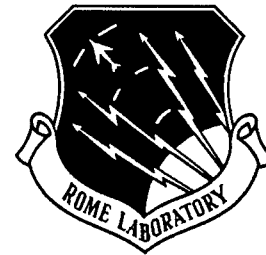


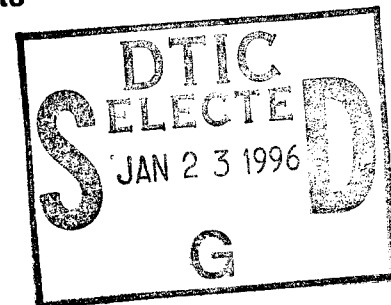
RL-TR-95-208
Final Technical Report
October 1995



DESIGN RECOVERY TECHNOLOGY FOR REAL- TIME SYSTEMS

The MITRE Corporation

Lester J. Holtzblatt, Richard Piazza, and Susan N. Roberts



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

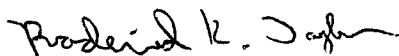
19960122 056

DATA CRAFTED BY SELECTED 1

**Rome Laboratory
Air Force Materiel Command
Rome, New York**

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR-95- 208 has been reviewed and is approved for publication.

APPROVED: 

RODERICK K. TAYLOR, Captain, USAF
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CA), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE October 1995		3. REPORT TYPE AND DATES COVERED Final Oct 92 - Jan 95	
4. TITLE AND SUBTITLE DESIGN RECOVERY TECHNOLOGY FOR REAL-TIME SYSTEMS				5. FUNDING NUMBERS C - F19628-89-C-0001 PE - N/A PR - MOIE TA - 74 WU - 01	
6. AUTHOR(S) Lester J. Holtzblatt, Richard Piazza, and Susan N. Roberts					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation Center for Air Force C3 Systems 202 Burlington Road Bedford MA 01730-1420				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CA 525 Brooks Rd Rome NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-95-208	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Roderick K. Taylor, Captain, USAF/C3CA/ (315) 330-2940					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Software maintainers typically rely on source code as the only completely reliable source of information on the software for which they are responsible. This process of trying to develop an understanding of the software by manually navigating through the code is extremely time consuming and error prone. This report describes a technology that automatically extracts information from the source code and presents the information in a comprehensible format. Further, the report details the reverse engineering tools and design recovery techniques used to accomplish these goals.					
14. SUBJECT TERMS Reverse engineering tools, Design recovery, Software analysis				15. NUMBER OF PAGES 80	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

TABLE OF CONTENTS

SECTION	PAGE
1 Introduction	1
1.1 Limitations of Reverse Engineering Tools	1
1.2 Overcoming the Limitations of Reverse Engineering Tools	2
1.2.1 Building Tools for Multiple Legacy Languages	2
1.2.2 Capturing Inter-task Communication in Legacy Systems	3
1.3 Background	4
2 The Language-Independent Model	5
2.1 Separating Parsing and Analysis	5
2.2 Creating an Intermediate Language-Dependent Representation	6
2.3 Language-Independent Analysis	6
2.4 Implementation Approaches	8
2.5 Language-Dependent Accessors	8
2.5.1 Analysis-Dependent Organization of Methods	9
2.5.2 Analysis-Independent Organization of Methods	10
2.5.3 Declarative Approach	10
2.5.4 Model Completion	11
2.6 Code Reuse	12
2.7 Conclusions	13
3 Task Flow Recovery	15
3.1 Determining the Task Called by RTOS	15
3.2 Determining the Calling Task	16
3.3 Implementing the Design Construct Recognition	17
3.4 Implementing Program Slicing	19
3.4.1 Intra-procedural Data Dependence Analysis	19
3.4.2 Inter-procedural Data Dependence Analysis	20
3.5 Evaluating Execution Threads Through a Program Slice	22
3.6 Extending M-CLUE to Recognize Other Services	23
3.7 Results	23
3.7.1 How the Results were Evaluated	23
3.7.2 Language Description for Graph Specification Language	24
3.7.3 Results	25

SECTION	PAGE
3.8 Limitations	28
3.8.1 Inherent Limitations of the Approach	28
3.8.2 Current Limitations of the Implementation	29
3.8.3 MCE Not Implemented According to MCE Documentation	31
3.9 Other Uses for the Tasking Graph Comparitor	31
3.9.1 Comparing Specifications to Implementation	31
3.9.2 Comparing Different Software Versions	32
3.9.3 Comparing for Debugging Test Flow Generation Software	32
3.10 Conclusions	32
4 Related Work	35
4 Summary	37
References	39
Appendix A	41

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

SECTION 1

INTRODUCTION

The software for many large military systems is maintained by DOD organizations that are responsible for both enhancing as well as correcting problems in these systems' software. The productivity of software maintenance organizations can be adversely affected by the considerable amount of time software maintainers spend simply trying to understand the software they are maintaining. Studies of the software maintenance process indicate that software maintainers, on the average, spend approximately one-half of their time developing an understanding of the software. One of the primary reasons for this is that the documentation and other formal descriptions of large software intensive military systems are often inadequate and unreliable. As a result, software maintainers typically rely on source code as the only completely reliable source of information on the software. This process of trying to develop an understanding of the software through manually navigating through the code is extremely time consuming and error prone. This situation has created a need for a technology that both automatically extracts information from the source code and presents this information in a comprehensible format.

1.1 LIMITATIONS OF REVERSE ENGINEERING TOOLS

Reverse engineering tools that extract certain aspects of the structure of a software system from source code are commercially available. Although these tools vary in the range of capabilities that they provide, most of them share a common core of capabilities. For example, all of the tools we have surveyed will display the calling hierarchy of a program, although they will vary in how this information is displayed. Many of the tools will also display information about the flow of control within individual procedures. Tools will also typically extract and display information concerning the structure and usage of data within a program. For example, tools will often generate reports concerning which procedures use or set particular variables. They will also typically display information concerning the structure of records, tables, or arrays used within a program.

These tools can provide maintainers insight into the structure of a program particularly when they are coupled with navigational aids. For example, using one of the family of tools available from Reasoning Systems, (Refine/C, Refine/Ada, Refine/Fortran, Refine/ Cobol), a software maintainer can interactively navigate through code by selecting different portions of code to view from a structure chart. A maintainer may also begin to gain insight into the potential impact of changes he plans to introduce into a program by using these tools to identify areas of the program that may be affected by making the change. In each of these

cases, reverse engineering tools may improve the productivity of a software maintainer both by providing the maintainer insight into the structure of a program and by making relevant portions of a program readily accessible to the maintainer.

In spite of the potential of these tools, two factors limit their utility for many DOD software systems. First, although reverse engineering tools are widely available for C, Fortran, Cobol, and Ada, relatively few tools are available for programming languages in which many older military applications were written (e.g., Jovial, CMS-2, and various assembly languages). In addition, different non-standard variants of common programming languages, for which no reverse engineering tools are commercially available, were often used.

Second, the utility of commercially available tools for many DOD software systems is limited by their ability to extract only information concerning the sequential execution of a computer program. Real-time military systems frequently consist of individual units of execution (tasks) that can operate concurrently on different processors or by interleaving their functioning on the same processor. These concurrent tasks typically exchange both control information as well as data through a variety of mechanisms. However, with the exception of tools that support Ada and its explicit tasking constructs, reverse engineering tools fail to capture information concerning the flow of information between tasks. As a result, these tools provide limited support for understanding the structure of real-time systems

1.2 OVERCOMING THE LIMITATIONS OF REVERSE ENGINEERING TOOLS

1.2.1 Building Tools for Multiple Legacy Languages

One factor limiting the general availability of reverse engineering tools for different legacy languages is the relative cost of building these tools. To the extent that these tools can rely on capabilities that can be reused to support multiple languages, the overall cost of building these tools should be significantly reduced.

Reverse engineering tools generally are not built to easily support porting analysis capabilities from one source language to another. One reason for this is that most reverse engineering tools tightly couple their two primary activities: parsing and analysis. To provide efficient analysis capabilities, a tool builder may decide to provide a fixed set of analysis reports and then optimize the tool to extract just the information needed for these reports. In this sort of design an explicit intermediate representation of the source program is usually not created thus saving greatly on memory costs. Furthermore, the information required for the reports can be extracted during the parsing phase and then populated into idiosyncratic data structures to support the required analyses. One tradeoff in such an approach is that flexibility to support unanticipated analysis capabilities is lost. By tightly

coupling parsing, analysis, and analysis optimization, the task of adding support for new languages or new analysis capabilities can be tantamount to a complete redevelopment activity.

Some tools may create an explicit intermediate representation of the source code. Doing this provides some flexibility in adding new analysis capabilities in that a clean interface to the source code information can be defined and the information itself persists after parsing. However, if this intermediate representation is syntax specific, then porting capabilities from source language to language will still be a major effort. In Section 2, we describe an approach to language-independent representation of source code that permits reuse of analysis capabilities within source languages of a particular family (e.g., 3GLs).

1.2.2 Capturing Inter-task Communication in Legacy Systems

As we noted above, reverse engineering tools generally fail to capture information concerning the interaction of concurrent tasks in a system. One of the primary reasons for this situation is that understanding design constructs relevant to the execution of concurrent tasks requires more than an implementation level understanding of the software [Biggerstaff, 1989]. The syntax of programming languages, particularly older legacy languages such as CMS-2 or Fortran, does not make constructs such as inter-task communication and task synchronization explicit. Instead, the inter-task behavior of a system often depends on the design of the specific operating system and the way in which the application code interacts with the operating system. Since reverse engineering tools only extract information that is represented explicitly in the syntax of the programming language, tools for sequential programming languages can only extract information concerning the sequential execution of individual tasks. These tools will fail to capture information concerning how these tasks interact.

However, people can often extract knowledge about how concurrent tasks interact from the source code of older systems, even though such information is not explicitly available in the syntax of a programming language. Extracting this information requires knowledge about the type of processing model used by the system software and how this processing model has been implemented in a particular system.

In addition to knowledge about the type of processing model used by a system, it is also necessary to understand the idiosyncratic techniques used by a system to implement these constructs. For example, although tasks may not be explicitly represented through syntactic constructs in the code, specific recurring patterns of code may be used to represent a task in a particular application. As a result, it may still be possible to recognize those specific portions of code that implement a particular task. Similarly, the specific actions through which these tasks communicate with each other may be implemented through particular types of calls to

the real-time operating system. Interpreting how specific tasks communicate with each other will depend on being able to interpret the meaning of these specific calls.

As can be seen, the ability of a person to manually extract extra-linguistic information from the source code of a program depends on his ability to use knowledge about how specific design constructs are implemented in the source code. Reverse engineering tools are not designed to make use of such meta-design knowledge. However, unless techniques are developed to make use of meta-design knowledge, reverse engineering tools will fail to extract more than the implementation level detail of a program. As long as tools can only provide limited visibility into the structure of a program, they will not be able to provide the insight required to understand the design of a real-time system. In Section 3, we describe an approach for using this meta-design information to support the extraction of task flow information from a system implemented in CMS-2.

1.3 BACKGROUND

In FY92 MITRE developed CLUE, CMS-2 Language Understanding Environment, a reverse engineering tool for CMS-2. CLUE was implemented on top of a commercially available reverse engineering development environment, Software Refinery from Reasoning Systems. Software Refinery consists of three components: Dialect, a parser generator that was used to build the CMS-2 parser; Software Refine, a programming language that was used to develop the CLUE reports; and InterVista, a graphical user interface builder that was used to develop the displays for these reports. Initially, CLUE consisted of two reports: a procedure calling hierarchy and a data set/use report. The research described in Section 2 extended the CLUE tool by incorporating several advanced analysis capabilities that were initially developed as extensions to Refine/C, a commercial reverse engineering tool for C. CLUE is available under a General Public Licence.

The research described in Section 3 was also built on top of CLUE, adding the capability to analyze the task flow behavior of a real-time system built in CMS-2, the Modular Control Equipment (MCE) system. CLUE, including the enhancements described in this report, have been made available to the Computer Support Squadron, Air Combat Command (CSS/ACC). This version of CLUE with its MCE specific extensions is known as M-CLUE. The CSS/ACC is responsible for maintaining MCE. M-CLUE has been integrated into the Theater Software Maintenance Environment (TSME) that has been procured by the MCE project to support maintenance activities of the CSS/ACC.

SECTION 2

THE LANGUAGE-INDEPENDENT MODEL

One of the goals of this work was to develop a tool that would be useful for many different legacy systems. Since legacy systems use a host of different source languages, it is important to discover ways to avoid creating a completely new tool for each language. This is necessary because commercial tools are probably not available for these languages due to their small non-commercial customer base. This is particularly true when assembly languages are involved. Therefore, since not much effort can be expended for each language, a method must be found that will make the development of a similar tool for another legacy language straightforward and inexpensive.

At first this problem might seem trivial. The answer might be summed up as “all you have to do is add another parser.” We found this solution to be naive based on our efforts to modify a tool in just this manner [Reubenstein et al., 1993]. Our approach is the use of a *language-independent model (LIM)*. A language-independent model can be the basis for implementing language-independent analysis capabilities that are not developed for a particular language but for a family of languages based on an abstract model. Analysis capabilities can then be written referencing this model without worrying about or depending upon the idiosyncrasies of any one language.

Our approach is predicated upon the ability to separate the issues of parsing and analysis. Using this approach a parser converts the source code into a language-dependent parse tree. We then use language-independent analysis capabilities to produce reports and higher-level views of the software. A small amount of language-dependent code must be written to access information from the parse tree, but the core code for the analysis capability is common to all languages. This makes the analysis capabilities transparent to the underlying language-dependent representation.

2.1 SEPARATING PARSING AND ANALYSIS

One approach to writing reverse engineering tools is to intermingle parsing and analysis. With this approach, it is straightforward to use parsing in a limited way to traverse the code to recognize constructs and gather information that is pertinent for a particular report. Therefore, there is often no internal representation of the source code.

A problem with this approach becomes clear when one tries to port an analysis capability to a new language. Since the analysis capability is intertwined with the parsing of the source

code and there is no common internal representation of the source code, it is difficult to reuse the analysis code. The analysis capability generally must be rewritten from scratch for the new language.

We feel that there is another problem with this approach. The intertwining of parsing and analysis is really a mixing of language, syntax, and semantics. The grammar productions used by a parser capture the syntax of a language, but individual productions do not necessarily correspond to the programming concepts of a language. Separation of these two concerns not only makes it possible to reuse much of the analysis code, but also frees the tool builder from needing to address subtle differences in languages when trying to write analysis capabilities.

2.2 CREATING AN INTERMEDIATE LANGUAGE-DEPENDENT REPRESENTATION

A language-dependent representation of the source code is created through parsing the code. Parsing takes the textual representation of software and converts it into an internal representation that captures the structure of the program as an Abstract Syntax Tree (AST). Parsers are usually constructed by an automatic parser generator that takes as input a formal description of the syntax of a language. We used a parser generator called DIALECT that is a part of a reverse engineering development environment known as Software Refinery from Reasoning Systems. The parser generated by DIALECT represents the nodes in the AST as objects that belong to a class of objects associated with the left-hand side of non-terminals of the appropriate grammar rule. The arcs in the AST are labeled. These labels represent part-whole syntactic relationships between the objects in the AST. For example, a node in the AST may represent a conditional statement. A conditional statement may be decomposed into several parts including a test condition, a then clause, and an else clause, each of which may be viewed as attributes of the conditional statement. Each of these parts are represented as objects. The arcs connecting the conditional statement to each of its parts are labeled with the corresponding names of the attributes they represent (i.e., test-condition, then-clause, else-clause).

2.3 LANGUAGE-INDEPENDENT ANALYSIS

Analysis routines operate on the contents of an AST by creating different views of the structure of the code. These views each reveal relationships between parts of the code that are not explicitly represented in the AST. These views include a procedure calling hierarchy, intra-procedural control flow, data flow, data set/use, and the task flows described in

Section 3. In addition, it is possible to perform analyses on top of these views to either compute statistical attributes of these views or to identify specific portions of a view that satisfy particular conditions. An example of a statistical analysis is the McCabe complexity metric which operates on a control flow graph to compute the number of independent test paths in a procedure. Such a measure provides an estimate of the understandability of a procedure and may be used to estimate the maintainability of the procedure [Banker et al., 1993]. Other analyses that identify relevant portions of a particular view include dead code detection and program slices. Dead code detection identifies portions of the code that will not be referenced. Dead code can be identified from the procedure calling hierarchy by identifying disconnected subgraphs of the hierarchy. A program slice identifies all of the statements that are used to compute the value of a particular variable used at a particular statement in the code. Program slices are derived from data flow and control flow graphs.

In order for analysis routines to be written in a language-independent manner, it is necessary for these routines to reference language constructs independent of the syntax of a specific programming language. For example, a language-independent implementation of a routine to generate control flow graphs needs to distinguish between types of statements represented in the AST. The types of statements it is looking for will fall into one of several predefined classes, including procedure-calls, while-loops, iteration-loops, conditionals, case-statements, and variable assignment statements. Each of these statement types represents language-independent constructs to which the analysis routine must make reference. Once a type of statement is identified the analysis routine will perform different actions depending on the type of statement identified. These actions must also reference language constructs in a language-independent manner. For example, if the analysis routine has identified a conditional block, then it needs to identify the test-condition, then-part, and else-part of the conditional. Each of these parts also represents a language-independent construct.

By implementing analysis routines that reference the AST through language-independent constructs, it is possible to port these routines from one programming language to another. In doing so, the language model used to generate the AST for a specific programming language is transparent to the analysis routines that are accessing information from the AST.

A language-independent model that identifies the set of language constructs that will be referenced by analysis routines cannot encompass all computer languages. There are several types of languages (e.g., algorithmic (third generation), functional, object-oriented, assembly). These language types have different semantic constructs and need different language-independent models. Since this project is focusing on legacy systems, we are developing two distinct language-independent models; one for third-generation languages and one for assembly languages.

2.4 IMPLEMENTATION APPROACHES

There are two approaches for implementing the LIM. In both approaches the analysis capability core code would be written making reference to the language-independent constructs like those described in Section 2.3. In the first and more obvious approach, the language-independent model is fully instantiated for the source code. In this approach a language-independent AST would be generated. As a result, all accessing of syntactic and semantic information from the AST would be exactly the same across different languages. Therefore, no extra code would need to be written to support the analysis capabilities. In a second approach, the language-independent attributes are computed via accessors which are specialized for each language. One can think of the two approaches to implementing language-independent analysis capabilities as being similar to the design decision often faced in object-oriented programming. In that situation there is a choice between storing or computing an object's state. Using a full LIM can be thought of as storing the state. The language-dependent accessor approach can be thought of as computing the object's state on an as needed basis.

Regarding the first LIM approach, there are two ways to generate a language-independent AST. First, it could be generated from a language-dependent AST. However, this would require two ASTs (one language-dependent and one language-independent). This would overwhelm the memory available on most processors. Another way is to generate a language-independent AST while parsing, possibly avoiding generating the language-dependent AST. However, this probably could not be done using the DIALECT tool in Software Refinery. Also, it would only reduce, but not eliminate, the amount of language-dependent code that was needed to generate the AST. Furthermore, access to the original source syntax, which is important for source code navigation, would be lost.

2.5 LANGUAGE-DEPENDENT ACCESSORS

Instead of computing a complete language representation of the AST, we implemented the second approach, which is the language-dependent accessor approach. Because of differences in the syntax and semantics of third generation languages, the DIALECT generated parsers for each language use different domain models and grammars, making the structure of their ASTs different. It is necessary to write some language-dependent code to specialize the accessors for each language. We will call these accessors *methods* – the colloquial name for a function that is specialized depending upon the type of its arguments.

These methods may be syntactic or semantic in nature. Some of the approaches below support only the implementation of syntactic methods. In one of the approaches the

accessors are generated automatically. In the rest of this section we will discuss the various approaches.

2.5.1 Analysis-Dependent Organization of Methods

When an analysis capability is written using the analysis-dependent approach, all methods are organized with the capability. When writing the analysis capability, a method stub is created for those parts of the code that are language-dependent. These method stubs are then implemented for each language in the tool suite. In this way, the core code of the analysis capability is reused. One disadvantage of this approach is that similar access routines written for one analysis capability will probably not be used in other analysis capabilities because they are not written for general use and are probably too idiosyncratic. One advantage is that the language-dependent code is only written on an as-needed basis.

The following is a list of some of the language-dependent methods which need to be written in order to support the language-independent data flow analyzer. Observe that the specification for each routine is straightforward and independent of the complexities of flow analysis.

- collect-primary-objects:
 given a code object this method returns call locations in the object's AST
- assignment-stmt?:
 true if the object represents an assignment or variable initialization
 (assignment statement, initialization)
- left-hand-side:
 the code object on the left-hand side of an assignment
- right-hand-side:
 the code object on the right-hand side of an assignment
- get-all-output-actual-arguments:
 all locations in a procedure call that will be assigned a value upon return
- get-global-variables:
 returns a list of global variables defined in the program
- global-variable?:
 true if the location represents a globally defined memory location
- pointer-variable?:
 true if the location represents a memory location that holds a pointer
- array-variable?:
 true if the location represents an array

2.5.2 Analysis-Independent Organization of Methods

In the analysis-independent approach the methods are organized around abstract language classes rather than a particular analysis capability. However, since Software Refinery does not explicitly support object-oriented programming, we have not defined explicit language-independent classes or class instances. Below are two examples – the method signatures associated with the abstract classes `procedure` and `procedure-call`.

`procedure`:

```
procedure-declarator(procedure) -> declarator
procedure-declarator?(any-type) -> boolean
procedure-called-by(procedure) -> set(procedure)
procedure-id-name(procedure) -> symbol
get-enclosing-procedure(any-type) -> procedure
root-of-procedure-hierarchy?(procedure) -> boolean
get-all-procedures(program) -> set(procedure)
get-procedure-with-name(symbol) -> procedure
get-root-of-procedure-hierarchy(set(procedure)) -> procedure
get-formal-arguments(procedure) -> set(any-type)
```

`procedure-call`:

```
procedure-call?(any-type) -> boolean
get-all-procedure-calls(any-type) -> set(procedure-call)
get-procedure-calls-with-name(symbol) -> set(procedure-call)
get-procedure-called(procedure-call) -> procedure
get-procedure-called-name(procedure-call) -> symbol
get-actual-arguments(procedure-call) -> set(any-type)
```

This approach has two advantages over the analysis-dependent method. First and foremost is the potential for greater reuse of code. Second is that this solution is more object-oriented, and therefore, a more locally understandable design.

2.5.3 Declarative Approach

Many of the language-dependent methods that are written are simply used to access the appropriate attribute in an abstract syntax tree. For instance, to obtain the “then” part of a conditional, which is a language-independent attribute, it is necessary to access a differently named slot in a C source AST as opposed to a CMS-2 source AST.

A more efficient way to write these slot accessors is to set up a mapping from the language-independent abstract class and attributes to the language-dependent class and attributes.

Once this mapping is constructed, it can be used by some utility routines to access/test for the language-dependent attribute/class given the name of the language-independent attribute/class.

A tool has been written to help build this mapping. It presents each language-dependent class and an example code fragment, together with a menu of the possible language-independent abstract classes. Once a class has been selected from the menu, the attributes are matched in a similar way.

In this example we look at a part of the map from two different languages of the same language-independent class – a case statement. A LIM case statement has two attributes: test and body.

In C the mapping is:

```
[switch-statement, lim::case, [< switch-expr, lim::test>, <switch-body, lim::body>]
```

This declaration states that switch-statement is C's case construct. The attribute switch-expr is where the case's test attribute is found and the switch-body is where the case's body attribute is found. Below is the declaration for the for-block, CMS-2's case construct.

In CMS-2 the mapping is:

```
[for-block, lim::case, [<for-expression, lim::test>, <value_block, lim::body>]
```

As in the C example, the language-dependent attributes are mapped to the language-independent attributes, test, and body, of a case statement.

2.5.4 Model Completion

Although third generation languages share abstract constructs, their syntax is often very different. Also, their attributes are sometimes represented implicitly. An example of this is an iteration loop. An iteration loop has several attributes: loop variable, initialization of loop variable, test, bump of loop variable, and loop body. In C these attributes are represented explicitly in the syntax and therefore the AST. This is an example of a C FOR statement.

```
for (i = 0; i < 10; i++)  
{  
    <some other statements>  
}
```

However, in CMS-2, some of the attributes are implicit in the syntax of the language (some are also optional); they are not explicitly represented in the source code. Therefore, they are

not in the parsed representation (AST). The following is an example of a CMS-2 VARY statement where many of the iteration loop attributes have been elided.

```
VARY I THRU 10  
  <some other statements>
```

Notice that the test, initialization, and bump are not expressed in this code fragment, but the semantics of the VARY statement indicate that the test is $i \leq 10$, the initialization is $i=0$, and the bump is $i = i+1$.

The language-independent analysis routines assume that all of the attributes of an iteration loop will be explicitly available. Additionally, it would be antithetical to the idea of language-independence to write special case code within the analysis capability core code to detect which attributes are pre-set. In the model completion approach, all of the ASTs are brought up to a “least common ancestor” by completing the model – making implicit attributes explicit by adding attributes. This is done in a preprocessing step, after parsing.

One can think of this approach as being a hybrid of the two main approaches. Because it is creating and storing information on the AST it has some of the flavor of the full LIM approach. However, this is done only to make implicit syntactic attributes explicit and not to replace language-dependent attributes with language-independent ones. One can think of the creation of the full LIM approach as the logical extension of model completion. However, since the language-dependent code needed to make the attributes explicit could be used during analysis, this approach is also similar to the language-dependent accessor approach.

2.6 CODE REUSE

The following table indicates the amount of code reuse that was achieved by the approaches in the previous section. Three different analysis capabilities are listed. The first column indicates the number of lines of language-independent core code. The second column indicates the number of lines which needed to be added to specialize the code so that it handles C. The third column indicates the number of lines which needed to be added to specialize the code so that it handles CMS-2. The fourth column indicates the approaches used.

Analysis Capability	Core Code	Code for C	Code for CMS-2	% Reuse	Approach Used ¹
Data Flow	1303	363	304	78.2	AD, AI
Control Flow	1137	123	266	90.5	D, MC
Orphans	56	-	-	100	AI

The language-dependent code for the control flow analysis capability was generated using the declarative approach. The reason why there are twice as many lines of code for CMS-2 is that the language domain model is twice as large as the domain model for C.

It is interesting to note that no extra code needed to be written to support orphan detection. This is because all of the language-dependent code required to support orphan detection had already been written to support data flow analysis. Since the code pertaining to procedures and procedure calls was organized using the analysis-independent approach, it was simple to locate and reuse the code when writing the orphan analysis.

2.7 CONCLUSIONS

A methodology has been developed to foster the reuse of analysis capabilities across reverse engineering tools for different languages. This methodology reduces the level of effort required to implement new analysis capabilities by partitioning analysis routines into language-dependent accessors and language-independent analyses. Once a language-independent analysis has been implemented, it can be ported to a new programming language by developing the accessors for that language. In addition, we are able to accomplish further reuse by allowing analysis routines to share the same accessors. As a result, once these accessors have been developed for one analysis routine, they can be reused to access the same information for another analysis routine.

To date, we have demonstrated considerable success in porting analysis routines between C and CMS-2. In addition, we have also begun to demonstrate reuse across analysis routines. Because of the lack of tools for assembly languages and the diversity of assembly languages used in DOD systems, this approach will have a significant payoff when applied to the development of assembly language reverse engineering tools. To address this need, we are developing a set of language-independent assembly language analysis reports in FY 94.

¹ Key for table: AD – analysis dependent, AI – analysis independent, D – declarative, MC – model completion.

SECTION 3

TASK FLOW RECOVERY

Although one of the goals of this work has been to develop techniques to recover the inter-task behavior of real-time systems in general, our initial efforts have centered on recovering this information from one system in particular, the Modular Control Equipment (MCE) system. MCE is a command and control system written in the Navy source language CMS-2. It also contains a relatively small amount of embedded assembler language. The assembler code is less than ten percent of the system and is predominately located in the real-time operating system (RTOS). The MCE software runs in a distributed, multiple CPU hardware environment. The software consists of 14 functional subprograms that comprise 44 CMS-2 modules. The software modules are distributed across the different CPUs. RTOS enables the software on different CPUs to communicate, sharing both data and control (task invocation).

Tasks in MCE are executable units within a module and are comprised of many different procedures. Tasks spawn a variety of actions on themselves or other tasks through procedure calls to RTOS. These actions include scheduling a task, terminating a task, or removing a previously scheduled task. We have focused primarily on developing techniques for determining which tasks schedule other tasks, although this approach can be extended to recover information regarding other types of operating system calls.

Determining the flow of tasks within MCE requires extracting information that is not directly available in the MCE source code. Extracting the task flows requires extracting the two primary pieces of information required to understand any task flow: who called a task and what task was called. Neither piece of information is explicitly represented in the source code. The following two sections will describe the overall strategy that was required to automatically extract this information from MCE.

3.1 DETERMINING THE TASK CALLED BY RTOS

The task scheduled by an RTOS call is uniquely determined by a set of arguments passed to RTOS by the RTOS call. These arguments identify a module and the task contained in that module. A module/task pair uniquely identifies a task in the MCE system.

In order to determine the task spawned by an RTOS call, it is necessary to determine the state of the two variables that uniquely identifies this task at the particular point in the program when an RTOS call is made. In some cases determining the value of these variables is relatively straightforward since these values are set once and then remain constant throughout the execution of that module. Other variables, however, are set multiple times within a

module. In these cases it is necessary to statically evaluate a portion of the program that determines the state of the variables.

When a variable is not preset, its state can be determined by identifying and evaluating the set of statements that may impact the value of that variable. Algorithms for identifying the minimal set of statements that may impact the state of a variable are known as program slicing [Weiser, 1984]. We implemented a program slicing algorithm to use when identifying the minimal set of statements impacting the module and task variables within an RTOS call. For our purposes, this technique assumed that the state of a module and task variable were completely determined within the scope of a task since the program slicing algorithm does not trace data dependencies across task boundaries. This assumption was valid for all but one module in the MCE system.

For any particular RTOS call, module and task variables may assume different values in different contexts. Because a program slice contains the set of all statements that may influence the state of a variable, only a subset of these statements may actually be executed under a particular context. In order to evaluate each program slice under each possible context, each syntactically possible execution thread within a module that may reach a designated RTOS call is evaluated. Each of these evaluations derives a distinct value for the module and task variables for the particular RTOS call. These values identify the maximal set of tasks that may be called by a specific source code RTOS call.

3.2 DETERMINING THE CALLING TASK

RTOS calls are made within the context of a particular task. A task is said to spawn some action on another task when an RTOS call is made within the context of that task. One of the difficulties in determining task flows is in determining which task spawned a particular action. This is because there exists no syntactic structure, such as a procedure, that corresponds to a task in CMS-2. Therefore, one cannot simply read the source code to determine the task containing a particular RTOS call.

Although no syntactic structure exists in CMS-2 that corresponds to a task, it is possible to determine which task spawns another through a call to RTOS by identifying the calling context for that call. Because this calling context is associated with a task, once the calling context is identified, the task that spawned this call can be identified. To do this, we needed to define a set of recognition rules that could be used to identify occurrences of MCE tasks. These recognition rules were based on our understanding of how tasks were implemented in MCE.

Tasks are activated in MCE when an "entry-procedure" for a module is called by RTOS. This entry-procedure is implemented by a CMS-2 construct known as a p-switch, which will pass control to one of a set of procedures depending on the value of the argument passed by RTOS to the p-switch. Each procedure to which an entry procedure can pass control represents the root procedure of a different MCE task. A task continues executing in MCE until the root procedure terminates.

To recognize the occurrences of tasks, we needed first to identify objects in the code that represented "entry-procedures" for modules. Once these entry-procedures were recognized the root procedures for each task could be identified by tracing through the p-switch. In order to recognize these entry-procedures we used information extracted from external documentation. Since this documentation was available in a structured format, we wrote a parser to extract the relevant information from the documentation. We used this information to select the p-switch in a file that functioned as the entry-procedure for a module. Once the entry-procedure for a module was identified we could identify the root procedure for each task contained in that module.

The identification of the root-procedure for each task provided the knowledge necessary for identifying the context of an RTOS call. As noted in Section 3.1, a specific RTOS call may be made within different contexts, resulting in different values for the module and task variable and hence spawning different tasks. Each calling environment contains a root procedure that corresponds to the calling task. Therefore, determining the task that spawned a new task requires determining the calling environment for a particular RTOS call passed a specific set of module/task values. This was done as part of evaluating each execution thread through a program slice.

3.3 IMPLEMENTING THE DESIGN CONSTRUCT RECOGNITION

The overall strategy for determining task flows required the implementation of a set of recognition rules that identified a small set of design constructs (e.g., tasks and modules) in the MCE code. This information was then supplemented with techniques for evaluating the states of specific variables in the code that identified the tasks spawned by a particular RTOS call. These evaluations required determining the program slice for the module and task variable in each RTOS call. This program slice was evaluated within the calling environment of each task within the module containing the RTOS call. This evaluation returned a value for the module and task variable together with the calling environment in which these values were computed. These values identified the task spawned by a particular RTOS call and the calling environment identified the task spawning the new task. In this section, we will describe the approach we implemented for recognizing design constructs in MCE source

code. In the following sections, we will describe how program slicing was implemented and how a program slice was evaluated to determine the task flow.

The purpose of design construct recognition is to identify instances of the design constructs implemented in a software system and their interrelationships. We created a domain model that identifies a small set of design constructs in the MCE code: tasks, modules, and a small set of events through which tasks interact with each other (e.g., tasking spawning). The current implementation hard-codes recognition rules for these design constructs. Each recognition rule creates an instance of an abstract design construct or determines the value of one of its attributes.

Because of the difficulty of recognizing these abstract design constructs from information contained solely in the source code, we implemented recognition rules that operated on both design documentation and the parsed representation of the source code. We were able to identify a portion of the on-line documentation for MCE which described each of the 44 modules of the system. For each module, the module's name and a list of tasks was listed. A list of files relevant to the module and the file that contained the entry procedure for the module were also identified.

Although this documentation was written in English, it was fairly structured. Thus, with a minimal amount of editing, we were able to automatically parse the documentation using a recursive-descent parser written in Refine. The parser automatically created module and task objects for each module and task identified in the documentation. The module and task names and the list of relevant files for each module were also set automatically during parsing.

After obtaining as much information about modules and tasks as possible from the documentation, we turned to the source code to complete the model. As noted in Section 3.1, each module is associated with an entry procedure. Because the documentation only identifies the name of the file containing a module entry procedure, we needed to find this procedure from the source code. This is done by generating the procedure calling hierarchy of the module. The module entry procedure is equivalent to the root procedure in the procedure calling hierarchy. To avoid orphan procedures, the root of the largest disconnected subgraph is used. As stated in Section 3.2, the module entry procedure contains a CMS-2 construct called a p-switch. The p-switch passes control to the entry procedure for a particular task depending upon the value of the p-switch variable. Therefore, from the p-switch we were able to determine the names of the task entry procedures for each of the tasks in that module.

Once modules, tasks, and their entry procedures have been recognized, it is possible to determine the behavior of each task by identifying and interpreting RTOS system calls used

by a task. Our domain model represents each of the events produced via an RTOS call and its associated attributes. We implemented event recognition algorithms that identify occurrences of these events.

The first step is to find all of the RTOS calls in the source code. This is easy to do by traversing the abstract syntax tree and testing for the name RTOS in each procedure call object encountered. The next step is to evaluate the value of the arguments used by RTOS to determine the task behavior the RTOS call represents. An RTOS call has two arguments; the type of the RTOS call and a table (a CMS-2 data structure) containing information for that type of call. The fields in the table vary depending on the type of RTOS call invoked. For example, if the RTOS call schedules a task, then the table includes two fields which contain the information necessary for the operating system to determine which task to schedule. For each RTOS call identified in the code, the first argument identifying the type of RTOS call is accessed and the appropriate event object is created to represent the event. Our algorithm then determines the task invoking this event and the values of designated fields in the table to determine the value of the event's attributes. This is done by computing and evaluating a program slice for the relevant fields in the table.

3.4 IMPLEMENTING PROGRAM SLICING

A program slice on some variable v , or set of variables, at statement n consists of those statements that contribute to the value of v just before statement n is executed. In the current implementation, we compute a program slice from a data flow graph.

A data flow graph is constructed by identifying a set of "reaching definitions" for each variable used in a program. Statement m is a reaching definition for variable v used by statement n when statement m defines the value of v actually used at n through some execution path. Note that a variable v in statement n may have several reaching definitions under different execution paths. n "backward depends" on m , and m "forward depends" on n . A backward (forward) program slice is computed on statement n by taking the transitive closure of all backward-depends (forward-depends) relations on statement n .

3.4.1 Intra-procedural Data Dependence Analysis

The first step needed to generate an intra-procedural data flow graph is to generate a control flow graph (CFG). A control flow graph for a procedure is a directed graph that contains an initial node which represents the entry point for a procedure and a final node which represents the procedure's exit point and a set of remaining nodes that each represent sequences of simple statements in the procedure represented by the CFG. Each edge in the graph represents a possible flow of control.

The next step in data dependency analysis is to identify the reaching definitions for each *location* used in a procedure. The term *location* is used instead of *variable* because it is necessary to keep track of arrays and data structures. Each node in a CFG is mapped to a set of locations defined and a set of locations used in the statement represented by a node. There exists a reaching definition between a definition and a use of a location if there is a path in the CFG between the node that contains the definition of the location and the node that contains the use of the location. Since a location may be defined multiple times within a procedure, there are many potential candidates for the definition that actually reaches a use of a location at a statement. It is possible for a location to have several reaching definitions because the definitions for that location are in the body of conditionals. However, a definition can also cancel another, eliminating the canceled definition as a reaching definition for all subsequent uses of that location.

3.4.2 Inter-procedural Data Dependence Analysis

We extended the concept of reaching definitions to take into account reaching definitions between statements contained in different procedures. Our extensions only consider reaching definitions contained within the scope of a single task. Reaching definitions that occur between tasks are not considered by our algorithm. Inter-procedural data flow analysis considers both global variables and parameter passing between procedures.

In order to support inter-procedural data flow analysis, the process is done in several steps. First, the control flow graph is generated. Second, within each procedure the definitions and uses of a location are computed. Third, reaching definitions are computed for all locations used in a procedure. Finally, the relations forward-depend and backwards-depends are computed. During this process the reaching definitions for global variables are found. Each step is done for all procedures, via a post-order traversal of the procedure calling hierarchy.

The inter-procedural reaching definitions for a global variable use can be found in one of three places: within the procedure (an intra-procedural reaching definition), in a procedure called by the procedure, or in a procedure which calls the procedure. Each of these are considered in order. First, the reaching definitions within the procedure are considered. If the reaching definition is a regular assignment statement, that statement is returned. Second, a procedure call contains the reaching definition, if the global variable was defined within that called procedure. The called procedure must be investigated to find the assignment statement which is the actual reaching definition. The intra-procedural information for all global variables defined within a procedure is summarized in the unique exit node of the CFG, so it is easy to access. Finally, if no definitions are found within the procedure or a called procedure, then all procedure that call this procedure must be investigated. This process is a recursive one, traversing the calling hierarchy as needed.

If the variable is a formal parameter, and is not defined within the procedure then the reaching definition must be the one implicit in parameter passing. Therefore, all of the calls to the procedure are the reaching definitions. This will work for call-by-value parameter passing. The issue of aliases (call by reference) or other parameter passing schemes have not been investigated.

An example of a program slice is below, with emphasis on the inter-procedural data flow.

```
int some_global_variable;

int p()
{
    int i = 0, z, x = 1, y = 2;
    z = x * y;
    if (i == 0)
    {
        i = 5;
    }
    else
    {
        i = 6;
    }
    t(0);
    z = i + some_global_variable;
    return z;
}

int t(x)
    int x;
{
    some_global_variable = 1;
}
```

The program slice for this program is:

```
p: i = 5
p: i = 6
p: t(0)
p: z = i + some_global_variable
p: return z
t: some_global_variable = 1
```

In this program the value of z at return z is computed using the previous statement, therefore the value of z depends upon *some_global_variable* and i . The value of i depends only on statements in procedure p . i is set conditionally, so both assignments appear in the slice. If conditionals were included in the slice, $i == 0$ would also appear. Note that even though it is easy to determine that i does equal 0 and, therefore, $i = 5$ is executed and not $i = 6$, both still appear in the slice because there is no way, in general, to determine statically what will happen when the program is executed. The value of *some_global_variable* is set in procedure t which is called by p . Therefore, the call to t and the assignment are included in the slice by using inter-procedural data flow analysis.

3.5 EVALUATING EXECUTION THREADS THROUGH A PROGRAM SLICE

Once a program slice is available it is possible to evaluate the slice to determine the possible values of the table fields used by an RTOS system call. Evaluation of a slice is made somewhat easier in CMS-2 because procedure invocation does not introduce a new scope. All variables in a CMS-2 program, including formal arguments of procedures, are global.

Inputs given to the evaluator are the variables of interest, a list of the variables for which values are requested, and the statement of interest, the statement in the slice for which the variable values should be evaluated. The execution of a slice is guided by a pre-order traversal of the procedure calling hierarchy. As it is traversed, each procedure that is encountered may contain some statements that are found in the slice. They are evaluated in the order in which they occur in the procedure (i.e., statements are sorted by line number) and their values are saved for use in other computations of the evaluator. When the statement of interest (the RTOS call in this case) is encountered, the values of the variables of interest are checkpointed. If the statement is encountered again, the values at that time are also checkpointed. Sets of checkpointed values together with the calling environment for each set are returned from the evaluator.

Given the value for the module/task pair and the calling environment, is it possible to compute the calling tasks and the called tasks of the RTOS call. A calling task is one whose entry procedure is contained in the calling environment. A called task is the one that corresponds to the module/task pair. If the module is the same as the one under investigation, the task number is an index into the module's entry procedure p-switch statement, which can be thought of as a list of all intra-module task entry procedures. If the module number corresponds to another module, then information from the documentation is used to determine the name of the task so that it can be displayed in the task flow graph or table.

3.6 EXTENDING M-CLUE TO RECOGNIZE OTHER SERVICES

Other services are provided via calls to the RTOS operating system in addition to spawning new tasks. As part of computing the automatically generated documentation, we also found these relationships. The method to find the calling task of the RTOS call is the same as outlined above. Different information is often sent to RTOS depending upon the service requested. These are also passed to RTOS as a slot in a data structure. The values of these slots can be computed via a program slice and evaluation in a way similar to the method used to discover the called module/task pair. The table below summarizes the other services that were evaluated, whether these services spawned tasks, and the type of information extracted regarding the service.

Services	Spawn Tasks	Other Information (Slots)
REGISTER	yes	period, delay
REGXTERNL	yes	period, delay
BUFREG	yes	period, delay
REMOVE	task removed from queue	
IOREQUES	no	file number
BUFIOREQ	no	file number
NOTIFY	no	message id
DMU	maybe	ddb-address

3.7 RESULTS

3.7.1 How the Results were Evaluated

As part of MITRE's support for the MCE system, each module was evaluated by hand and written documentation including a tasking graph was prepared. These manually generated graphs identified each task within a module, the set of tasks called by each task, and other service calls made to RTOS by a task.

In order to validate the approach for generating tasking/service graphs that was discussed above, it was necessary to compare the automatically generated graphs with these manually generated ones. The Tasking Graph Comparitor was developed to automatically accomplish this task.

The results of comparing the automatically generated graphs with manually generated graphs are discussed in Section 3.7.3. Many of the inconsistencies found were caused by errors in

the manually generated documentation, and not by the algorithm to produce the automatically generated graphs. These discrepancies are discussed in Section 3.8 and Appendix A.

3.7.2 Language Description for Graph Specification Language

To be compared, automatically generated and manually generated graphs must have a common representation. There must be some way to specify the manually generated graphs in a form that is readable by the machine so that it can be put into that common representation. For this reason a graph specification language was developed. This language is used to enter the relationship between tasks and other RTOS services. Essentially, it is a way to specify each arc of the graph. Together a set of the individual specifications of these relationships is used to form the graph.

For instance, to specify the arc in the ATA graph between task ATA5 and DSE9 the following specification would be used:

```
(def-rtos-info
  :caller-module ATA
  :caller-task-number 5
  :called-module DSE
  :called-task-number 9)
```

To specify that a particular io-request was made by a module, the following specification would be used:

```
(def-rtos-info
  :type *rtos-io-request-call
  :caller-module SCB
  :caller-task-number 11
  :file-number 50)
```

A full description of the specification language is below.

```
<rtos-info> -> (def-rtos-info <type>
  :caller-module <module-name>
  :caller-task-number <integer>
  <attribute-list> )
```

```

<type> -> *rtos-register-call | *rtos-remove-call | *rtos-io-request-call |
          *rtos-buffer-register-call | *rtos-register-external-call |
          *rtos-notify-call | *rtos-buffer-io-request-call |
          *dmu-read-call | *dmu-write-call | *dmu-distribute-call |
          <empty>

```

```

<module-name> -> ATA | CMB | CMC | ....

```

```

<attribute-list> -> <attribute> <attribute-list> | <attribute>

```

```

<attribute> -> :called-task <module-name> :called-task-number <integer> |
              :notify <integer> |
              :ddb-address <integer> |
              :file-number <integer> |
              :delay <integer> |
              :line-number <integer> |

```

3.7.3 Results

In the table below we list the actual results of comparing the manually and automatically generated graphs for each module of the MCE system. The columns contain data about the two different sets of information which were manually and automatically generated, respectively. The number in common indicate the number of relationships in the intersection of these two sets. The *recall* statistic measures the number of relationships found and how correct they were. Recall does not consider relationships that are not in the intersection. In this way it measures the number of false negatives. The lower the recall, the fewer correct relationships were found.

$$\text{Recall} = \frac{\text{number of relationships found by both methods}}{\text{number of relationships generated manually}}$$

Precision indicates the ratio of correctly found relationships over all of the relationships found. In this way it indicates the percentage of extra relationships found (i.e., the number of false positives). If the precision is low that means a high percentage of extra relationships were found.

$$\text{Precision} = \frac{\text{number of relationships found by both methods}}{\text{number of relationships generated automatically}}$$

Table 3-1. Results Comparing Manually Generated and Automatically Generated Graphs

Module	Number in Manually Generated Graph	Number in Automatically Generated Graph	Number in Common	Recall	Precision
ATA	19	21	18	94.7	85.7
CMB	classified				
CMC	classified				
CMD	classified				
CME	classified				
CMF	classified				
CMG	classified				
CMU	49	49	45	91.8	91.8
CXF	classified				
CXM	classified				
CXU	13	13	10	76.9	76.9
DMA	9	6	6	66.7	100.0
DMB	9	9	8	88.9	88.9
DRA	41	42	41	100.0	97.6
DSA	not run				
DSB	non-standard				
DSC	not run				
DSD					
DSE	37	57	29	78.4	50.9
PMA	56	56	54	96.4	96.4
RTH	non-standard				
SCB	165	140	83	50.3	59.3
SIA	not run				
SIB	120	211	109	90.8	51.7
SMA	24	24	17	70.8	70.8
SMB	19	19	19	100.0	100.0
SMD	5	5	5	100.0	100.0
SME	36	36	29	80.6	80.6
SMF	44	36	16	54.5	66.7
SMG	18	18	17	94.4	94.4
SRC	35	33	33	94.3	100.0
SRD	39	40	39	100.0	97.5
SRE	25	22	22	88.0	100.0
SRF	non-standard				
SRG	9	9	9	100.0	100.0

Table 3-2. Optimal Results Based on "Corrected" Manually Generated Graphs

Module	Number in Manually Generated Graph	Number in Automatically Generated Graph	Number in Common	Recall	Precision
ATA	21	21	20	95.24	95.24
CMB	classified				
CMC	classified				
CMD	classified				
CME	classified				
CMF	classified				
CMG	classified				
CMU	49	49	49	100.0	100.0
CXF	classified				
CXM	classified				
CXU	13	13	13	100.0	100.0
DMA	6	6	6	100.0	100.0
DMB	9	9	9	100.0	100.0
DRA	42	42	42	100.0	100.0
DSA					
DSB	non-standard				
DSC					
DSD					
DSE	42	57	34	80.9	59.6
PMA	56	56	56	100.0	100.0
RTH	non-standard				
SCB					
SIA	not run				
SIB	141	211	132	93.6	62.6
SMA					
SMB	19	19	19	100.0	100.0
SMD	5	5	5	100.0	100.0
SME	36	36	28	77.8	77.8
SMF	44	36	29	65.9	80.6
SMG	18	18	18	100.0	100.0
SRC	35	33	33	94.3	100.0
SRD	40	40	40	100.0	100.0
SRE	25	22	22	88.0	100.0
SRF	non-standard				
SRG	9	9	9	100.0	100.0

This leads to an optimal average performance for a module of 99.7% recall and 98.6% precision.

The average performance for a module is 85.9% recall and 85.5% precision. Each discrepancy was investigated to determine if there was an error in the manually generated graphs or in the algorithm to compute the graphs automatically. Appendix A explains each of the discrepancies in detail. As noted, a number of these discrepancies were due to errors in the manually generated graph. The following table expresses the optimal results that would be obtained if the manually generated graphs were correct.

3.8 LIMITATIONS

In this section we will enumerate reasons for the incorrect results when the automatically generated graphs produced the wrong result. In Appendix A we will take a detailed look at all inconsistencies and relate them to these limitations.

3.8.1 Inherent Limitations of the Approach

1. The program slicer is a static tool. It cannot take into consideration dynamic information. Additionally, the data flow analyzer does not currently take other control flows into consideration, i.e., exception or error handling.
2. The algorithms generating task flows assumed that all variables are set within the module. This is for the most part true, with some exceptions. One example where this occurs is in module CXU (see SCA in Appendix A for another example).

Missing from Automatically Generated Graph:

CXU4 IOREQUES 2084
 CXU5 IOREQUES 2084
 CXU6 IOREQUES 2084

Not in Manually Generated Graph:

CXU4 IOREQUES 2052
 CXU4 IOREQUES 2052
 CXU4 IOREQUES 2052

All of these discrepancies are caused by the same code.

CMU.2573 (CXUSMD)

SET CXURTIO(0, BUSADDR) TO 2052 + ((GCUMARK(0, DCUADDR) *
 32))

CMU.2574 (CXUSMD)

RTOS INPUT IOREQUES, CORAD(CXURTIO) OUTPUT GPOINTER

The problem is caused by the fact that GCUMARK(0, DCUADDR) is never set to anything within the module.

3.8.2 Current Limitations of the Implementation

Some of the results are incorrect because of current limitations of the implementation. Most of these limitations could be eliminated by enhancing the implementation of the program slicer, CMS-2 source code evaluator, or the task flow graph generator. The following enumerates these limitations in more detail.

1. The slicing algorithm does not include test statements in a conditional as part of a slice because of the computational explosion that may result from including these statements. As a result, a slice may not be correctly evaluated. For example, consider the following MCE code:

```
SMAGEN.2050 (SMARDSCN):  
  IF GVMODE EQ MODIFYSM  
    OR GVMODE EQ PRINTSM  
  THEN BEGIN  ''READ SCENARIO SMAGEN''$  
    SET IDDMSRD(0,MODULE) TO SMAGENMN  ''SMAGEN MODULE''$  
    SET IDDMSRD(0,TASK) TO SMARDCTN  
    END      ''READ SCENARIO SMAGEN''$  
  ELSE BEGIN  ''LOAD SCENARIO SMGSWP''$  
    SET IDDMSRD(0,MODULE) TO SMGSWPMN  ''SMGSWP MODULE''$  
    SET IDDMSRD(0,TASK) TO SMGRDCTN  
    END      ''LOAD SCENARIO SMGSWP''$
```

The statements

```
SET IDDMSRD(0,MODULE) TO SMAGENMN  ''SMAGEN MODULE''$  
SET IDDMSRD(0,MODULE) TO SMGSWPMN  ''SMGSWP MODULE''$
```

are both part of the slice for IDDMSRD(0,MODULE) since either statement will affect its value.

The problem arises when evaluating this slice. Slices do not include information regarding the context in which a statement should be executed intra-procedurally. In actuality, only one branch of the conditional is executed at any time, which means that a slice may include two or more different contexts. However, since the evaluator simply uses line numbers to establish the order of execution of the statements within a procedure, all but one execution path will be to be obscured.

This is because the execution of the last assignment to the variable will overwrite the other previous assignments.

In the case of the example, the evaluator will overwrite the statement:

```
SET IDDMSRD(0,MODULE) TO SMGSWPMN  ''SMGSWP MODULE''$
```

with the statement:

```
SET IDDMSRD(0,MODULE) TO SMAGENMN  ''SMAGEN MODULE''$ .
```

When computing the tasking graph we must investigate all contexts. The missing conditional information causes the evaluator to miss at least one context.

To remedy this limitation, it is necessary to enhance the evaluator so that it uses the control flow graph to order the statements, starting a new context when encountering conditionals.

2. Statements controlling iteration are not included in the slice or handled by the evaluator. In MCE source code, many IO requests are made within a loop. For example, in the following code:

```
VARY VSUINDX FROM 0 THRU SMNRDR - 1
IF GTMOCU(VSUINDX,RDRNO) EQ RADARIFF THEN ''VSU sim'ing Rdr''
BEGIN
  SET IDRIOPKT(0,BUSADDR) TO VSUINDX + 567 ''Set Logical
  RTOS INPUT IOREQUES, CORAD(IDRIOPKT) OUTPUT DUMMY
END
END ''VARY Loop''
```

only the RTOS call that would have been executed the first time through the loop will be evaluated. Therefore, many IO requests are missing

3. Impossible Paths. Certain control flow paths are not possible since they may depend on a contradictory set of conditions. Since tests within a conditional are not evaluated these impossible paths are still evaluated, leading to the identification of task calls that would never occur.
4. Array/Pointer References. In CMS-2 an address of a table can be set to a variable via the language construct CORAD. Since we do not perform data flow analysis on what is essentially a pointer, we miss certain relationships. An example of this occurs in module SIB.

```

SIBATO.13818 (SIBMSACT)
    SIBREAD INPUT CORAD(IDMRPMRC)
SIBATO.13945 (SIBREAD)
    SET CORAD(IPACKET) TO PCKTADDR - the formal parameter
SIBATO.13961 (SIBREAD)
    DMUREAD INPUT CORAD(IPACKET) ...

```

Since we don't know what the value is of IPACKET (it is IDMRPMRC), we cannot determine the task/module pair of this call.

3.8.3 MCE Code not Implemented According to the MCE Documentation

The degree to which the automatically generated graphs accurately capture the correct relationships is dependent upon how well the actual MCE source code adheres to the documented design of the MCE system. This is necessary because many of the heuristics built into the tool depend upon this.

In particular, in modules DSB, RTH, and SRF the p-switches in the module entry procedures are non-standard. Therefore, the heuristics don't work very well, and we get incomplete results.

3.9 OTHER USES FOR THE TASKING GRAPH COMPARITOR

The Comparitor can also be used for additional purposes. It could be used to compare the tool's results to specifications of the real time system created during the design phase. It can be used to compare different versions of the system to insure consistency. Lastly, it can be used in software debugging of our tool. Below we discuss all of these uses in more detail.

3.9.1 Comparing Specification to Implementation

The Tasking Graph Comparitor can be used to compare the automatically generated graphs to a specification of the task flow which was produced during the design phase (although such documentation might be less detailed than the manually generated graphs that MITRE prepared). In this way, a comparison between the design and the implementation would be possible. A situation similar to this could arise if new modules were added to the MCE system. In general, such a comparitor could be an important part of a software forward engineering tool set.

3.9.2 Comparing Different Software Versions

The MCE system is undergoing constant changes during its maintenance phase. New versions of the system are produced. Under the assumption that the higher level abstractions of the software do not change much from version to version, the tasking relationships should remain relatively unchanged. It would be useful to compare the tasking graphs from the old and the new versions to see the differences. This would be a good way to focus on the impact of the changes made in the new version. Although certain differences would be correct, other unexpected differences could indicate a software bug.

3.9.3 Comparing for Debugging Test Flow Generation Software

Lastly, the Comparitor was very helpful when debugging the task flow graph generator. First, it was useful to compare the automatically generated graphs to the manually generated ones so that one could focus in on where problems with the code could be. Additionally, the current results could be compared to previous results so that it could be determined that a bug fix to correct the graph of one module did not cause an incorrect graph to be generated for another module. In this way, changes to the software could be made with more confidence.

3.10 CONCLUSIONS

The real-time tasking tool implemented for MCE demonstrates that a significant level of design recovery can be obtained when a small amount of design knowledge regarding a system is encoded into a powerful set of tools and then applied in an analysis across the entire system. The current implementation hard codes recognition rules for a small set of MCE relevant objects (e.g., tasks and modules) and a small set of MCE relevant events (e.g., tasking spawning via RTOS calls). The notion of “objects” of interest in a program and “events” of interest that relate objects to each other is a generic way to view the design of a software system. We are building a framework that supports the recognition of objects and events in an attempt to capture what can be termed the architecture [Shaw, 1989; Perry and Wolf, 1992] of a software system. This recognition framework supports the specification of recognition rules for object and event types (versus hard coded system specific rules) and provides powerful visualization facilities for the set of events recognized in a program [Harris, 1995].

The techniques used in this system are all static analysis techniques and thus are inherently limited by the degree to which static analysis can be used to evaluate run time behavior. We have implemented a program slicing technique that enables static evaluation of program

values where feasible. This capability is currently limited in its ability to deal with name aliasing (an intractable problem), but we are continuing to increase its abilities to resolve aliases and statically evaluate resulting program slices.

By applying powerful program analysis capabilities in concert with recognition rules derived from some basic system design knowledge a significant level of system design recovery can be achieved. The information is derived directly from the source code and traceable back to that source code. As software baselines change, design recovery can be reapplied to produce current design information, yielding a form of "living" documentation that can reliably aid in program maintenance and understanding.

SECTION 4

RELATED WORK

Our approach is influenced by work that recognizes the organizational power of software architecture, techniques for program understanding and concept formation, and the enabling technology of reverse engineering.

Software architectures as described by [Shaw, 1989; Perry and Wolf, 1992] provide the organizational patterns that drive the real-time design recovery system. In this case, the architecture we analyzed and recognized was a fairly simple real-time system architecture in which modules and tasks are the relevant architectural objects and the key architectural event is the spawning of tasks.

Traditional program understanding work has focused on cliché-based recognition [Rich and Wills, 1990; Harandi and Ning, 1990]. Generally, this is a bottom-up recognition approach in which the program is matched to a set of pre-defined plans/clichés from a library. Recognition is based on a precise data and control flow match which indicates that the recognized source component is precisely the same as the library template. Our approach is more of a top-down hypothesis driven recognition approach coupled with bottom-up recognition rules. Our recognition rules do not require algorithmic equivalence of the plan and the source being matched, rather they are based on source code level events in the code. Quilici [1993] also explores a mixed top-down, bottom-up recognition approach using traditional plan definitions. The style of source code event-based recognition rules is also exemplified in [Kozzaczynski, et al., 1992; Engberts et al., 1991] which demonstrates a combination of precise control and data flow relation recognition and more abstract code event recognition.

Design recovery work, such as DESIRE [Biggerstaff, 1989] relies on: externally supplied cues regarding program structure, modularization heuristics, manual assistance, and informal information. Informal information and heuristics can also be used to reorganize and automatically refine recovered software designs as in the modularization tool described in [Schwanke, 1991]. In general, design recovery tools need to take advantage of extra-linguistic information such as that provided in our design recovery rules.

SECTION 5

SUMMARY

CLUE, a reverse engineering tool that MITRE developed for analyzing CMS-2 programs, was enhanced through the addition of several advanced analysis capabilities. The enhanced CLUE, known as M-CLUE, will be integrated with other maintenance tools as part of TSME, an automated maintenance environment for MCE. The analysis capabilities developed for M-CLUE should provide software maintainers improved visibility into the structure of the MCE software, allowing them to more quickly localize errors in the code and to more thoroughly assess the potential impact of changes in the code on other system components. The effect of these capabilities on the activities of software maintainers will be assessed as part of an evaluation of the effectiveness of the TSME environment.

A number of capabilities were developed for Refine/C and CLUE using a language-independent model (LIM). The LIM approach to developing capabilities allows the separation of parsing and analyses in the capabilities, thus enabling the capabilities to be ported to reverse engineering workbenches for different source languages with a minimal amount of effort. A LIM has been developed for third generation languages. This approach will be particularly useful for developing reusable capabilities for reverse engineering assembly languages.

A task flow analysis capability was developed as part of M-CLUE to automatically extract information from MCE source code regarding the behavior of concurrent tasks in MCE. Although this capability recovers this information from just the MCE system, the approach we used to develop this capability can be generalized to recover the inter-task behavior of real-time systems in general. In particular, our success in implementing this capability demonstrates that a significant level of design recovery can be obtained when a small amount of design knowledge regarding a system is encoded into a powerful set of tools and then applied in an analysis across the entire system.

LIST OF REFERENCES

[Banker et al., 1993]

R. Banker, S. Datar, C. Kemerer, and D. Zweig
Software Complexity and Maintenance Costs
CACM, November 1993, Volume 36-11

[Biggerstaff, 1989]

T. Biggerstaff
Design Recovery for Maintenance and Reuse
IEEE Computer, July 1989

[Engberts, et al., 1991]

A. Engberts, W. Kozaczynski, and J. Ning
Concept Recognition-based Program Transformation
Proceedings of the 1991 IEEE Conference on Software Maintenance, 1991

[Harris et al., 1995]

D. R. Harris, H. B. Reubenstein, A. S. Yeh
Reverse Engineering to the Architectural Level
Proceedings of the Working Conference on Reverse Engineering, July, 1995

[Harandi and Ning, 1990]

M. Harandi and J. Ning
Knowledge-based Program Analysis
IEEE Software, 7(1), 1990

[Kozaczynski and Ning, 1989]

W. Kozaczynski and J. Ning
SRE: A Knowledge-based Environment for Large-scale Software Re-engineering Activities
Proceedings of the 11th International Conference on Software Engineering, 1989

[Perry and Wolf, 1992]

D. Perry and A. Wolf
Foundations for the Study of Software Architectures
ACM Software Engineering Notes, 17(4), 1992

[Quilici, 1993]

A Hybrid Approach to Recognizing Program Plans
Proceedings of the Working Conference on Reverse Engineering, May 1993

[Rich and Wills, 1990]

C. Rich and L. Wills
Recognizing a Program's Design: A Graph Parsing Approach
IEEE Software, 7(1), 1990

[Reubenstein et al., 1993]
H. Reubenstein, R. Piazza, and S. Roberts
Separating Parsing and Analysis in Reverse Engineering Tools
Proceedings of the Working Conference on Reverse Engineering, May 1993

[Schwanke, 1991]
M. Schwanke
An Intelligent Tool for Re-engineering Software Modularity
Proceedings of the 13th International Conference on Software Engineering, 1991

[Weiser, 1984]
M. Weiser
Program Slicing
IEEE Transactions on Software Engineering, SE-10(4), July, 1984

APPENDIX A

TASKING GRAPH DISCREPANCIES

The following is a detailed description of the discrepancies between the manually generated documentation and the automatically generated documentation. For each module we have summarized the differences in a discrepancy report and discuss the reasons for the differences. Differences that cannot be accounted for are in bold in the discrepancy report.

ATA:

Discrepancy Report

Missing:

ATA01 -> ATA05

Extra:

ATA01 -> ATA05 delay 5.0 sec

ATA05 -> ATA05 delay: uneval expr

ATA06 -> ATA06

- ATA5 -> ATA5 is in the graph, but doesn't have a delay. The slice which indicates that there should be a delay is:

```
IF (DAYDIFF GT 1)
THEN SET TIMEDIF TO 100
ELSE BEGIN
  IF (DAYDIFF EQ 1)
  THEN SET TIMEDIF TO MINUTE + 1440 - TMINDAY
  ELSE BEGIN
    SET TIMEDIF TO MINUTE - TMINDAY
    IF (DAYDIFF LT 0) OR (TIMEDIR LT 0)
    THEN SET TIMEDIF TO 0
  END
  IF (TIMEDIF GT 0) AND (TIMEDIR LT SKEDTIME)
  THEN SET SHEDTIME TO TIMEDIR

ATAFS.4796 (ATAMON)
  SET SKEDTIME TO 30 $
ATAFS.4975 (ATAMON):
  SET IDMNRO(0,DELAY) TO SKEDTIME * 120 - 60 $
ATAFS.4978 (ATAMON):
  RTOS INPUT REGISTER, CORAD(IDMNRO) OUTPUT DUMMY $
```

- ATA1 -> ATA5, does not have a delay in the documentation, but one was found. The relevant code is:

```
ATAFS.4351 (ATAPOS):
    SET IDMNRO(0,DELAY) TO 10
ATAFS.4353 (ATAPOS)
    RTOS INPUT REGISTER, CORAD(IDMNRO) OUTPUT DUMMY $
```

- ATA6 -> ATA6 was found by is not in the documentation. The relevant code is:

```
ATAFS.1782
    TABLE IDPENT H 4 1 ''TASK 6 REQUEST PACKET'' $
ATAFS.1783
    FIELD MODNUM I 8 U 0 15 P ATAFSMN ''MOD NUMBER'' $
ATAFS.1784
    FIELD TSKNUM I 8 U 0 7 P 6 ''TASK NUMBER'' $

ATAFS.5314 (ATACAL)
    RTOS INPUT REGISTER, CORAD(IDPENT) OUTPUT DUMMY $
```

CMU

Discrepancy Report

Missing:

```
CMU6 -> CMU6
CMU7 -> CMU7
CMU17 -> CMU14 REMOVE
CMU18 -> CMU9 delay 500msec
```

Extra:

```
CMU6 -> CMU6 delay 500 msec
CMU6 -> CMU7
CMU18 -> CMU9 REMOVE
CMU18 -> CMU14 REMOVE
```

- CMU6 -> CMU6 is missing, but CMU6 -> CMU6 delay 500msec was generated. The correct answer is from the automatically generated graph. The relevant code is:

```
CMU.5519 (CMUG)
    CMUSKED INPUT REGISTER, CMUMN, AUTOINP, DELAY500, ZERO, NODATA
```

where AUTOINP is a equals tag with value task 6 and CMUG is the task entry procedure for task 6.

- CMU7 -> CMU7 is missing, but CMU6 -> CMU7 was generated. In task 7 (CMUH) there is no call to CMU7, however, in task 6 (CMUG) there is the following call:

```
CMU.5531 (CMUG)
  CMUSKED INPUT REGISTER, CMUMN, MSGINTRP, NODELAY, ZERO, NODATA
```

where MSGINTRP is an equals tag with value task 7, and CMUG is the task entry procedure for task 6

- CMU18 -> CMU9 delay 500msec is missing, but CMU18 -> CMU9 REMOVE was generated. The correct answer is from the automatically generated graph. The relevant code is:

```
CMU.13215 (CMUMCAN)
  CMUSKED INPUT REMOVE, CMUMN, AUTOOUTP, NODELAY, ZERO, NODATA
```

where AUTOOUTP is a equals tag with value task 9 and CMUMCAN is the task entry procedure for task 18.

- CMU17 -> CMU14 REMOVE is missing, but CMU18 -> CMU14 REMOVE was generated. In task 17 (CMUH) there is no call to CMU14, however, in task 18 (CMUMCAN) there is the following call:

```
CMU.13219 (CMUMCAN)
  CMUSKED INPUT REMOVE, CMUMN, RAUTOUTP, NODELAY, ZERO, NODATA
```

where RAUTOUTP is an equals tag with value task 14, and CMUMCAN is the task entry procedure for task 18.

CXU

Discrepancy Report

Missing:

```
CXU4 IOREQUES 2084
CXU5 IOREQUES 2084
CXU6 IOREQUES 2084
```

Extra:

CXU4 IOREQUES 2052
CXU4 IOREQUES 2052
CXU4 IOREQUES 2052

All of these discrepancies are caused by the same code.

```
CMU.2573 (CXUSMD)
    SET CXURTIO(0, BUSADDR) TO 2052 + ((GCUMARK(0, DCUADDR) * 32))
CMU.2574 (CXUSMD)
    RTOS INPUT IOREQUES, CORAD(CXURTIO) OUTPUT GPOINTER
```

The problem is caused by the fact that GCUMARK(0, DCUADDR) never set to anything within the module.

DMA

Discrepancy Report

Missing:

DMA3 IOREQUES *UNDEFINED*
DMA3 IOREQUES 515
DMA3 CHGFILA

The task entry procedure for DMA3 is the same as the one for DMA3, therefore, in the interpretation of the task flow graph generator task DMA3 really doesn't exist (it looks for the first match of task entry procedure in the list of tasks).

DMB

Discrepancy Report

Missing:

DMB5 -> DMB5 IOREQUES 2054

Extra:

DMB5 -> DMB5 IOREQUES 0

The task entry procedure for DMB5 is DMBCYMSM. This directly calls RTOS without any intervening calls to other procedures. Since the ddb address field of the table(s) passed to RTOS are is not pre-set, the ddb-address is 0. The ddb-address of those tables are set in DMB4, which often calls DMB5, but this is only known because of the task flow, so this data flow dependency cannot be used.

DRA

Discrepancy Report

Missing:

DRA00 -> DMUDISTR 224
DRA17 -> DMUDISTR 224

Extra:

DRA12 -> DRA7

- The relationship DRA12 -> DRA7 is found, but is not in the manually generated graph. The relevant code is:

```
DRARAR.524 (IDRT7RQ)
  FIELD MODNUM I 8 U 0 15 P DRARARNM
  FIELD TSKNUM I 8 U 0 7 P DRATKP

DRARAR.7582 (DRAWRTK)
  DRATKOFF
DRARAR.6357 (DRATKOFF)
  RTOS INPUT REGISTER, CORAD(IDRT7RQ) OUTPUT DUMMY
```

where DRATKP is task 7 and DRAWRTK is the task entry procedure for task 12.

DSA

Discrepancy Report

- DSA3 -> CXM17 DMUDISTR 32 is called from DSADISTR, which is called from DSARSTRT which is DSA3.
- DSA5 -> CXM17 DMUDISTR 32 is called from DSADISTR, which is called from DSAPOINT which is DSA5.

DSD

Discrepancy Report

Missing:

DSD4 -> CXC7
DSD6 -> CXC7
DSD7 -> CXC7
DSD9 -> CXC7
DSD10 -> CXC7

Extra:

DSD4 -> CXC9
DSD6 -> CXC9
DSD7 -> CXC9
DSD9 -> CXC9
DSD10 -> CXC9

- DSDx -> CXC(M)7 should be DSDx -> CXC(M)9. There are no references to task number 7 associated with the CXC(M) module. All references to CXC(M) are associated with module number 9. An example of the relevant code is:

```
DSDHP.14739 (DSDRETRL)
    SET GMVRMN TO CXMXTLMN
DSDHP.14740 (DSDRETRL)
    SET GMVTASK TO 9
DSDHP.14741 (DSDRETRL)
    CSGPACK INPUT GMEEXR, 1
CSDSG1.1274 (CSGPACK)
    DSGPKBF
CSDSG1.1029 (DSGPKBF)
    SET IDBFSRQ(0, MNUM) TO GMVRMN
CSDSG1.1030 (DSGPKBF)
    SET IDBFSRQ(0, TASK) TO GMVTASK
CSDSG1.1035 (DSGPKBF)
    RTOS INPUT BUFREGST, CORAD(IDBFSRQ) OUTPUT GMVPTR
```

DSE

Discrepancy Report

Missing:

DSE3 -> IOREQUES 16
DSE4 -> IOREQUES 16
DSE5 -> SMC4
DSE5 -> IOREQUES 16
DSE6 -> SRF7
DSE6 -> IOREQUES 16
DSE7 -> IOREQUES 16
DSE8 -> IOREQUES 16

Extra:

DSE3 ->
DSE3 -> IOREQUES 0
DSE3 -> IOREQUES 18
DSE3 -> IOREQUES 54
DSE3 -> DSB5
DSE4 ->
DSE4 -> DSB5
DSE4 -> IOREQUES 0
DSE4 -> IOREQUES 18
DSE4 -> IOREQUES 54
DSE5 ->
DSE5 -> IOREQUES 0
DSE5 -> IOREQUES 18
DSE5 -> IOREQUES 54
DSE6 -> DSB5
DSE6 -> IOREQUES 0
DSE6 -> IOREQUES 17
DSE6 -> IOREQUES 18
DSE6 -> IOREQUES 54
DSE7 -> DSB5
DSE7 -> IOREQUES 0
DSE7 -> IOREQUES 18
DSE7 -> IOREQUES 54
DSE8 -> DSB5
DSE8 -> IOREQUES 0
DSE8 -> IOREQUES 18
DSE8 -> IOREQUES 54

- DSE6 -> SRF7 should be called, but the module/task pair is set in a conditional, and because of the limitations of the evaluator, the else part of the code is executed in the same context, and overwrites the settings for SRF7. The relevant code is:

```
DSEAP.16575 (DSEDROP)
    IF IDADMSUB(0, FRMTNM) EQ FTM2038
DSEAP.16577 (DSEDROP)
    SET GMVTASK TO 7
DSEAP.16578 (DSEDROP)
    SET GMVRMN TO SRFTMMN
DSEAP.16586 (DSEDROP)
    SET GMVTASK TO 5
DSEAP.16587 (DSEDROP)
    SET GMVRMN TO DSADPMN
```

- DSE5 -> SMC4 should be called, but the module/task pair is set in a conditional, and because of the limitations of the evaluator, the else part of the code is executed in the same context, and overwrites the settings for SRF7. The relevant code is:

```
DSEAP.14504 (DSEWCFF)
    IF DSEORDRQ(QINDX, PARM8) GT 0 THEN
DSEAP.14508 (DSEWCFF)
    SET GMVRMN TO SMCDLKMN
DSEAP.14509 (DSEWCFF)
    SET GMVTASK TO 4
DSEAP.14514 (DSEWCFF)
    SET GMVRMN TO CXMCTLMN
DSEAP.14515 (DSEWCFF)
    SET GMVTASK TO 9
```

- DSE3 -> DSB5, DSE4 -> DSB5, DSE6 -> DSB5, DSE7 -> DSB5, DSE8 -> DSB5 is generated via:

```
CSDSG1.946 (DSGCHECK)
    SET GMVRMN TO DSBPMN
CSDSG1.947 (DSGCHECK)
    SET GMVTASK TO 5
CSDSG1.949 (DSGCHECK)
    CSGALLOC INPUT GMEANS
CSDSG1.560 (CSGALLOC)
    SET IDRTSRQ(0,MNUM) TO GMVRMN
CSDSG1.561 (CSGALLOC)
    SET IDRTSRQ(0,TASK) TO GMVTASK
CSDSG1.958 (DSGCHECK)
    CSGSEND INPUT GMEANS
CSGSG1.695 (CSGSEND)
    RTOS INPUT REGISTER, CORAD(IDRTSRQ) OUTPUT GMVPTR
```

DSECHNG (DSE6), DSESAVIT (DSE7) and DSERSTRT (DSE3) all call DSECLERA which calls DSGCHECK. DSEPSMTO (DSE8) calls DSECLEARO which calls

DSEPLMOD, which calls CSGPACK, which calls DSGCHECK. DSEQALRT (DSE4) calls DSECAPAL which calls CSGPACK.

- DSE3 ->, DSE4 ->, DSE5 -> are all caused by calls to CGSEND. One argument, GMEANS, is passed to CGSEND, and within that function, as REGISTER call or an IOREQUES call is made, depending upon the value of the argument. When the call to CGSEND is to perform the IOREQUES, then no module/task pair is set. If conditionals were in the slice then these extra relationships could be avoided.

PMA

Discrepancy Report

Missing:

PMA5 -> PMA6
PMA11 -> RTH7

Extra:

PMA5 -> PMA6 delay 500 msec
PMA10 -> CMG6

- PMA5 -> PMA6 is missing but PMA5 -> PMA6 delay 500 msec was found. The relevant code is:

```
PMA.12019 (PMACCSR):  
    SET IDRTSR(0, PMDLY) TO 1  
PMA.12020 (PMACCSR):  
    RTOS INPUT REGISTER, CORAD(IDTSR), OUTPUT PMDUM
```

- PMA10 -> CMG6 should be in the manually generated documentation based on the following code. PMAES is task 10.

```
PMA.13812 (PMAES)  
    PMARESET  
PMA.18186 (PMARESET)  
    SET IDRTSR(0, PMNM) TO CMGDMN  
PMA.18187 (PMARESET)  
    SET IDRTSR(0, PMTM) TO CMGLNKST  
PMA.18191 (PMARESET)  
    RTOS INPUT REGISTER, CORAD(IDRTSR), OUTPUT PMDUM
```

- PMA11 -> RTH7 doesn't appear to be necessary. PMAPCM (task 11) makes no RTOS/DMU calls. It calls only one procedure, PMACVTME, which calls no other procedures, including RTOS/DMU.

RTH

Discrepancy Report

Missing:

SIB

Extra:

SIB1 -> SIB4 delay 60 sec
SIB3 -> DMUREAD 51
SIB3 -> SIB4 delay 60 sec
SIB3 -> SIB17 DMUDISTR 51
SIB3 -> SIB17 DMUWRITE 22
SIB3 -> SIB19 DMUREAD 245
SIB3 -> SIB22 DMUREAD 245
SIB3 -> WCA10
SIB4 -> SIB4 delay 500 msec
SIB4 -> DMUDISTR 51
SIB10 -> DSD7
SIB11 -> SIB20 REMOVE
SIB17 -> SIB17 DMUWRITE 23
SIB18 -> CMU8
SIB23 -> DSE4
SIB23 -> DMUDISTR 51
SIB23 -> SIB4 delay 60 sec
SIB23 -> SIB17 DMUDISTR 51
SIB23 -> SIB17 DMUWRITE 22
SIB23 -> SIB19 DMUREAD 245
SIB23 -> SIB22 DMUREAD 245

- SIB1,3 -> SIB4 should have a delay of 60 secs. The relevant code is:

```
SIBATO.8602 (TABLE IDTOPERD)
      FIELD MN ... SIBATOMN
SIBATO.8603 (TABLE IDTOPERD)
      FIELD TN ... CTNPER
SIBATO.8608 (TABLE IDTOPERD)
      FIELD TIMED...120
SIBATO.11700 (SIB)
      SIBATAOP USING IDTSK
SIBATO.11845 (SIBPSTIN)
```

```

        RTOS INPUT REGISTER CORAD(IDTOPERD) OUTPUT DUMMY
SIBATO.12502 (SIBRSTIN)
        SIBFILAC
SIBATO.12743 (SIBFILAC)
        SIBFILCU
SIVATO.11965 (SIBFILCU)
        RTOS INPUT REGISTER CORAD(IDTOPERD) OUTPUT DUMMY

```

- SIB3 -> DMUDISTR 51 was found and should be in the graph. The relevant code is:

```

SIBATO.6355 (TABLE IDMWGATO)
        FIELD FILEID ... CNGATAOG
SIBATO.11700 (SIB)
        SIBATAOP USING IDTSK
SIBATO.12502 (SIBSTRT)
        SIBFILAC
SIBATO.12743 (SIBFILAC)
        SIBFILCU
SIBATO.11962 (SIBFILCU)
        SIBSACT
SIBATO.13824 (SIBSACT)
        SIBPERWT
SIBATO.15355 (SIBPERWT)
        SET IDMWGATO(0, MN) TO 0
SIBATO 15356 (SIBPERWT)
        SET IDMWGATO(0, TN) TO 0
SIBATO.15357 (SIBPERWT)
        DMUDISTR INPUT CORAD(IDMWGATO) ...

```

- SIB3 -> SIB17 DMUWRITE 22 was found and should be in the graph. The relevant code is:

```

SIBATO.6415 (TABLE IDMWPLET)
        FIELD FILEID ... CNSIBLTF
SIBATO.6425 (TABLE IDMWPLET)
        FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
        SIBATAOP USING IDTSK
SIBATO.12502 (SIBSTRT)
        SIBFILAC
SIBATO.12743 (SIBFILAC)
        SIBFILCU
SIBATO.11962 (SIBFILCU)
        SIBSACT
SIBATO.13824 (SIBSACT)
        SIBPERWT
SIBATO.15348 (SIBPERWT)
        SET IDMWPLET(0, TN) TO CTNPWRTC - equal tag for 17
SIBATO.15349 (SIBPERWT)
        DMUREAD INPUT CORAD(IDMWPLET) ....

```

- SIB3 -> SIB17 DMUWRITE 51 was found and should be in the graph. The relevant code is:

```
SIBATO.6355 (TABLE IDMWGATO)
    FIELD FILEID ... CNGATAOG
SIBATO.6359 (TABLE IDMWGATO)
    FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBSTRT)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13832 (SIBSACT)
    SET IDWGATO(0, TN) TO CTNPWRTC - equal tag for 17
SIBATO.13833 (SIBSACT)
    DMUDISTR INPUT CORAD(IDMWGATO)....
```

- SIB3 -> SIB19 DMUREAD 245 was found and should be in the graph. The relevant code is:

```
SIBATO.6598 (TABLE IDMRFCUR)
    FIELD FILEID .... MNSIBMRF
SIBATO.6602 (TABLE IDMRFCUR)
    FIELD MN .... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBSTRT)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13824 (SIBSACT)
    SIBPERWT
SIBATO.15345 (SIBPERWT)
    SIBMAINC
SIBATO.15508 (SIBMAINC)
    SET IDMRFCUR(0, TN) TO CTNHIGHC
SIBATO.15511 (SIBMAINC)
    DMUREAD INPUT CORAD(IDMRFCUR)....
```

- SIB3 -> SIB22 DMUREAD 245 was found and should be in the graph. The relevant code is:

```
SIBATO.6598 (TABLE IDMRFCUR)
    FIELD FILEID .... MNSIBMRF
SIBATO.6602 (TABLE IDMRFCUR)
    FIELD MN .... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
```

```

SIBATO.12502 (SIBSTRT)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11949 (SIBFILCU)
    SET IDMRFCUR(0,TN) TO CTNFUTUR - equal tag for 22
SIBATO.11950 (SIBFILCU)
    DMUREAD INPUT CORAD(IDMRFCUR) OUTPUT...

```

- SIB3 -> WCA10 was found and should be in the graph. The relevant code is:

```

SIBATO.7619 (TABLE TRTOALMS)
    FIELD MN ... WCACTLMN
SIBATO.7620 (TABLE TRTOALMS)
    FIELD TN ... CTNWCMSG - equal tag for 10
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12466 (SIBSTRT)
    RTOS INPUT REGISTER, CORAD(TRTOALMS) OUTPUT GPOINTER

```

- SIB4 -> SIB4 delay 500 msec was found and should be in the graph, which was a different delay than that in the manually generated graphs. The relevant code is:

```

SIBATO.8665 (TABLE IDREPERD)
    FIELD MN .... SIBATOMN
SIBATO.8666 (TABLE IDREPERD)
    FIELD TN ... CTNPER
SIBATO.8671 (TABLE IDREPERD)
    FIELD TIMED ... 1
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.13016 (SIBPER)
    RTOS INPUT REGISTER, CORAD(IDREPERD) OUTPUT DUMMY

```

- SIB4 -> DMUDISTR 51 was found and should be in the graph. The relevant code is:

```

SIBATO.6355 (TABLE IDMWGATO)
    FIELD FILEID ... CNGATAOG
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12806 (SIBMSACT)
    SIBPERWT
SIBATO.15355 (SIBPERWT)
    SET IDMWGATO(0, MN) TO 0
SIBATO 15356 (SIBPERWT)
    SET IDMWGATO(0, TN) TO 0
SIBATO.15357 (SIBPERWT)
    DMUDISTR INPUT CORAD(IDMWGATO)...

```

- SIB10 -> DSD7 was found and should be in the graph. The relevant code is:

```
SIBATO.7997 (TABLE IDTOSTDU)
    FIELD MN ... DSDHPMN
SIBATO.7998 (TABLE IDTOSTDU)
    FIELD TN ... 7
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.16583 (SIBNMSA2)
    RTOS INPUT REGISTER, CORAD(IDTOSTDU) OUTPUT GPOINTER
```

- SIB11 -> SIB20 REMOVE was found and should be in the graph. The relevant code is:

```
SIBATO.9017 (TABLE IDTOCLRP)
    FIELD MN ... SIBATOMN
SIBATO.9018 (TABLE IDTOCLRP)
    FIELD TN ... CTNCLRPC
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.17633 (SIBDISP)
    RTOS INPUT REMOVE, CORAD(IDTOCLRP)...
```

- SIB12 -> WCA10 was found and should be in the graph. The relevant code is:

```
SIBATO.7619 (TABLE TRTOALMS)
    FIELD MN ... WCACTLMN
SIBATO.7620 (TABLE TRTOALMS)
    FIELD TN ... CTNWCMSG - equal tag for 10
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.18601 (SIBDSCMB)
    SIBDSPRC
SIBATO.18354 (SIBDSPRC)
    SIBEDDSR
SIBATO.19285 (SIBEDDSR)
    SIBMNDAT
SIBATO.20146 (SIBMNDAT)
    SIBDEACT
SIBATO.14624 (SIBDEACT)
    SIBNDALT
SIBATO.17353 (SIBNDALT)
    SIBALRTD
SIBATO.13595 (SIBALRTD)
    RTOS INPUT BUFREGST, CORAD(TRTOALMS)....
```

- SIB17 -> SIB17 DMUWRITE 23 was found and should be in the graph. The relevant code is:

```
SIBATO.6454 (TABLE IDMWPMNU)
    FIELD FILEID.... CNSIBMSN
SIBATO.6463 (TABLE IDMWPMNU)
    FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.15804 (SIBPWRTC)
    SET IDMWPMNU(0, TN) TO CTNPWRTC
SIBATO.15805 (SIBPWRTC)
    DMUWRITE INPUT CORAD(IDMWPMNU)....
```

- SIB17 -> SIB17 DMUWRITE 51 was found and should be in the graph. The relevant code is:

```
SIBATO.6355 (TABLE IDMWGATO)
    FIELD FILEID ... CNGATAOG
SIBATO.6359 (TABLE IDMWGATO)
    FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.15813 (SIBPWRTC)
    SET IDWGATO(0, TN) TO CTNPWRTC - equal tag for 17
SIBATO.15814 (SIBPWRTC)
    DMUDISTR INPUT CORAD(IDMWGATO)....
```

- SIB18 -> CMU8 was found and should be in the graph. The relevant code is:

```
SIBATO.7190 (TABLE IDCOMMTS)
    FIELD MN ... CMUMN
SIBATO.7191 (TABLE IDCOMMTS)
    FIELD TN ....8
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.26581 (SIBMTSMG) - TASK 18
    RTOS INPUT REGISTER, CORAD(IDCOMMTS) OUTPUT GPOINTER
```

- SIB23 -> DSE4 was found and should be in the graph. The relevant code is:

```
SIBATO.8411 (TABLE IDTOAEMD)
    FIELD MN ... DSEAPMN
SIBATO.8412 (TABLE IDTOAEMD)
    FIELD TN ... 4
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12806 (SIBRSTCP) - TASK 23
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
```



```

SIBATO.11962 (SIBFILCU)
    SIBMSACT
SIBATO.13800 (SIBMSACT)
    SIBAEWG INPUT SIBLTFF(NDX,MSNNDX)
SIBATO.27164 (SIBAEWG)
    RTOS INPUT BUFREGST, CORAD(IDTOAEMD) OUTPUT GPOINTER

```

- SIB23 -> SIB4 delay 60 sec was found and should be in the graph. The relevant code is:

```

SIBATO.8602 (TABLE IDTOPERD)
    FIELD MN ... SIBATOMN
SIBATO.8603 (TABLE IDTOPERD)
    FIELD TN ... CTNPER
SIBATO.8608 (TABLE IDTOPERD)
    FIELD TIMED...120
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.11845 (SIBRSTCP)
    RTOS INPUT REGISTER CORAD(IDTOPERD) OUTPUT DUMMY
SIBATO.12502 (SIBRSTIN)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIVATO.11965 (SIBFILCU)
    RTOS INPUT REGISTER CORAD(IDTOPERD) OUTPUT DUMMY

```

- SIB23 -> DMUDISTR 51 was found and should be in the graph. The relevant code is:

```

SIBATO.6355 (TABLE IDMWGATO)
    FIELD FILEID ... CNGATAOG
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12806 (SIBRSTCP)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13824 (SIBSACT)
    SIBPERWT
SIBATO.15355 (SIBPERWT)
    SET IDMWGATO(0, MN) TO 0
SIBATO 15356 (SIBPERWT)
    SET IDMWGATO(0, TN) TO 0
SIBATO.15357 (SIBPERWT)
    DMUDISTR INPUT CORAD(IDMWGATO)...

```

- SIB23 -> SIB17 DMUDISTR 51 was found and should be in the graph. The relevant code is:

```
SIBATO.6355 (TABLE IDMWGATO)
    FIELD FILEID ... CNGATAOG
SIBATO.6359 (TABLE IDMWGATO)
    FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBRSTCP)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13832 (SIBSACT)
    SET IDMWGATO(0, TN) TO CTNPWRTC - equal tag for 17
SIBATO.13833 (SIBPERWT)
    DMUREAD INPUT CORAD(IDMWGATO) ....
```

- SIB23 -> SIB17 DMUDISTR 22 was found and should be in the graph. The relevant code is:

```
SIBATO.6415 (TABLE IDMWPLET)
    FIELD FILEID ... CNSIBLTF
SIBATO.6425 (TABLE IDMWPLET)
    FIELD MN ... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBRSTCP)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13824 (SIBSACT)
    SIBPERWT
SIBATO.15348 (SIBPERWT)
    SET IDMWPLET(0, TN) TO CTNPWRTC - equal tag for 17
SIBATO.15349 (SIBPERWT)
    DMUREAD INPUT CORAD(IDMWPLET) ....
```

- SIB23 -> SIB19 DMUREAD 245 was found and should be in the graph. The relevant code is:

```

SIBATO.6598 (TABLE IDMRFCUR)
    FIELD FILEID .... MNSIBMRF
SIBATO.6602 (TABLE IDMRFCUR)
    FIELD MN .... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBRSTCP)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11962 (SIBFILCU)
    SIBSACT
SIBATO.13824 (SIBSACT)
    SIBPERWT
SIBATO.15345 (SIBPERWT)
    SIBMAINC
SIBATO.15508 (SIBMAINC)
    SET IDMRFCUR(0, TN) TO CTNHIGHC
SIBATO.15511 (SIBMAINC)
    DMUREAD INPUT CORAD(IDMRFCUR)....

```

- SIB23 -> SIB22 DMUREAD 245 was found and should be in the graph. The relevant code is:

```

SIBATO.6598 (TABLE IDMRFCUR)
    FIELD FILEID .... MNSIBMRF
SIBATO.6602 (TABLE IDMRFCUR)
    FIELD MN .... SIBATOMN
SIBATO.11700 (SIB)
    SIBATAOP USING IDTSK
SIBATO.12502 (SIBRSTCP)
    SIBFILAC
SIBATO.12743 (SIBFILAC)
    SIBFILCU
SIBATO.11949 (SIBFILCU)
    SET IDMRFCUR(0,TN) TO CTNFUTUR - equal tag for 22
SIBATO.11950 (SIBFILCU)
    DMUREAD INPUT CORAD(IDMRFCUR) OUTPUT...

```

SCA

Discrepancy Report

Missing:

SCA1 -> SCA5,6,7,8 REMOVE
SCA4 -> SCA5,6,7,8,9 REMOVE
SCA4 -> SCA5,6,7,8,9 DELAY 3,5,10
SCA4 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA4 -> SCA 15,16,17,18,19 REMOVE
SCA5 -> SCA6,7,8,9 REMOVE
SCA5 -> SCA5,6,7,8,9 DELAY 3,5,10
SCA5 -> SCA10 REMOVE
SCA5 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA5 -> SCA 16,17,18,19 REMOVE
SCA5 -> IOREQUES 2091
SCA6 -> SCA5,6,7,9 DELAY 3,5,10
SCA6 -> SCA10 REMOVE
SCA6 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA6 -> SCA 15,17,18,19 REMOVE
SCA6 -> IOREQUES 2091
SCA7 -> SCA5,6,7,8,9 REMOVE
SCA7 -> SCA5,6,7,8,9 DELAY 3,5,10
SCA7 -> SCA10 REMOVE
SCA7 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA7 -> SCA 15,16,18,19 REMOVE
SCA7 -> IOREQUES 2091
SCA8 -> SCA5,6,7,9 REMOVE
SCA8 -> SCA5,6,7,8,9 DELAY 3,5,10
SCA8 -> SCA10 REMOVE
SCA8 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA8 -> SCA 15,16,18,19 REMOVE
SCA8 -> IOREQUES 2091
SCA9 -> SCA5,6,7,9 REMOVE
SCA9 -> SCA5,6,7,8 DELAY 3,5,10
SCA9 -> SCA10 REMOVE
SCA9 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA9 -> SCA 15,16,18 REMOVE
SCA9 -> IOREQUES 2091
SCA10 -> SCA5,6,7,9 REMOVE
SCA10 -> SCA5,6,7,8 DELAY 3,5,10
SCA10 -> SCA10 REMOVE
SCA10 -> SCA15,16,17,18,19 DELAY 3,5,10
SCA10 -> SCA 15,16,19 REMOVE
SCA10 -> IOREQUES 2091
SCA11 -> SCA5,6,7,8,9 REMOVE
SCA11 -> SCA5,6,7,8 DELAY 3,5,10

SCA11 -> SCA10 REMOVE
 SCA11 -> SCA15,16,17,18,19 DELAY 3,5,10
 SCA11 -> SCA 15,17,18,19 REMOVE
 SCA11 -> IOREQUES 2091
 SCA12 -> SCA5,6,7,8,9 REMOVE
 SCA12 -> SCA5,6,7,8 DELAY 3,5,10
 SCA12 -> SCA10 REMOVE
 SCA12 -> SCA15,16,17,18,19 DELAY 3,5,10
 SCA12 -> SCA 15,16,17,18,19 REMOVE
 SCA12 -> IOREQUES 2091
 SCA13 -> SCA5,6,7,8,9 REMOVE
 SCA13 -> SCA5,6,7,8 DELAY 3,5,10
 SCA13 -> SCA10 REMOVE
 SCA13 -> SCA15,16,17,18,19 DELAY 3,5,10
 SCA13 -> SCA 15,16,17,18,19 REMOVE
 SCA13 -> IOREQUES 561
 SCA13 -> IOREQUES 2091
 SCA14 -> SCA5,6,7,8 DELAY 3,5,10
 SCA15 -> SCA 6,7,8,9 REMOVE
 SCA15 -> SCA5,6,7,8 DELAY 3,5,10
 SCA15 -> SCA10 REMOVE
 SCA15 -> SCA 15,16,17,18,19 DELAY 3,5,10
 SCA15 -> SCA 16,17,18,19 REMOVE
 SCA15 -> IOREQUES 2091
 SCA16 -> SCA 5,7,8,9 REMOVE
 SCA16 -> SCA5,6,7,8 DELAY 3,5,10
 SCA16 -> SCA10 REMOVE
 SCA16 -> SCA 15,16,17,18,19 DELAY 3,5,10
 SCA16 -> SCA 15,17,18,19 REMOVE
 SCA16 -> IOREQUES 2091
 SCA17 -> SCA 5,6,8,9 REMOVE
 SCA17 -> SCA5,6,7,8 DELAY 3,5,10
 SCA17 -> SCA10 REMOVE
 SCA17 -> SCA 15,16,17,18,19 DELAY 3,5,10
 SCA17 -> SCA 15,16,18,19 REMOVE
 SCA17 -> IOREQUES 2091
 SCA18 -> SCA 5,6,7,9 REMOVE
 SCA18 -> SCA5,6,7,8 DELAY 3,5,10
 SCA18 -> SCA10 REMOVE
 SCA18 -> SCA 15,16,17,18,19 DELAY 3,5,10
 SCA18 -> SCA 15,16,17,19 REMOVE
 SCA18 -> IOREQUES 2091
 SCA19 -> SCA 5,6,7,8 REMOVE
 SCA19 -> SCA5,6,7,8 DELAY 3,5,10
 SCA19 -> SCA10 REMOVE
 SCA19 -> SCA 15,16,17,18,19 DELAY 3,5,10
 SCA19 -> SCA 15,16,17,18 REMOVE
 SCA19 -> IOREQUES 2091

SCA20 -> SCA 5,6,7,8,9 REMOVE
SCA20 -> SCA5,6,7,8 DELAY 3,5,10
SCA20 -> SCA10 REMOVE
SCA20 -> SCA 15,16,17,18,19 DELAY 3,5,10
SCA20 -> SCA 15,16,17,18,19 REMOVE

Extra:

SCB

Discrepancy Report

Missing:

Extra:

SCB18 -> SRD6

- SCB18 -> SRD6 should be included because of the following code:

```
SCBINT.9022
    FIELD MODNUM ....SRDIDMN
SCBINT.9023
    FIELD TASKNUM ...6
SCBINT.24787 (SCBSMINP)
    SCBRIUIN
SCBINT.21826 (SCBRIUIN)
    SCBWNDOW
SCBINT.22918 (SCBWNDOW)
    SCBGOTWE
SCBINT.23112 (SCBGOTWE)
    SCBSNDWE
SCBINT.23221 (SCBSNDWE)
    RTOS INPUT REGISTER, CORAD(IDSRVWDW) OUTPUT DUMMY
```

- SCB11 -> DSC4 should have been found, but because of the conditional (FOR stmt)
SCB11 -> DSA5 was found instead. The relevant code is:

```
SCBINT.14218 (SCBDSPOU)
    FOR IDSUBHRD(0, FRMTNMBR)
SCBINT.14220 (SCBDSPOU)
    SET IDDSPACM(0,MODULE) TO DSCRPMN
SCTINT.14221 (SCBDSPOU)
    SET IDDSPACM(0,TASK) TO 4
SCBINT.14224 (SCBDSPOU)
    SET IDDSPACM(0,MODULE) TO DSADPMN
```

```

SCBINT.14225 (SCBDSPOU)
    SET IDDSPACM(0, TASK) TO 5
SCBINT.14230 (SCBDSPOU)
    RTOS INPUT REGISTER, CORAD(IDDSPACM) OUTPUT DUMMY

```

• SCB3,4,5,6,7,8,11,12,13,14,15,18 -> SMF6 appears instead of SCB4,5,6,7,8,11,12,13,14,15,18. This is because of the conditional problem in program slicing. The following code is contained in procedure SCBSIMOU.

```

@DP15.12.2FOR IDSMH(0,MSSGTYPE) $
@DP15.12.2    BEGIN RDRCMDS  '' radar command''$
@DP15.12.2        SET CORAD(IDSMHD) TO CORAD(BUFFRADR)  $
@DP15.12.2        FOR IDSMHD(0,CMDTYPE) $
@DP15.12.2            BEGIN EMCONON,EMCONOFF,IFFMODES $
@DP15.12.2                SET IDSMPK(0,TASK) TO 4 $
@DP15.12.2                SET IDSMPK(0,MODULE) TO SMFRIGMN $
@DP15.12.2            END $
@TR22939.2        BEGIN 15  ''FILTER,RDRREG,MIGSTS''  $
@DP15.12.2            ''FILTER DESIGNATION,RADAR REGISTRATION,MIG
STATUS REQUEST'\
'
@TR22939                IF IDSMHD(0,SUBCODE) EQ 3
@B/CS0006                THEN BEGIN  ''request MIG status''$
@TR22939                    SET IDSMPK(0,TASK) TO 4 $
@TR22939                    SET IDSMPK(0,MODULE) TO SMFRIGMN $
@TR22939                END $
@TR22939                ELSE BEGIN  ''1''  $
@TR22939                    IF IDSMHD(0,SUBCODE) EQ 2
@B/CS0006                    THEN BEGIN  ''filter designation cmd''$
@DP15.12.2                        SET IDSMPK(0,TASK) TO 9 $
@DP15.12.2                        SET IDSMPK(0,MODULE) TO SMETDGMN $
@DP15.12.2                    END $
@DP15.12.2                ELSE
@B/CS0006                    SET FNDIT TO TRUE  ''radar correction cmd.'' $
@TR22939                END ''ELSE BEGIN 1''  $
@TR2293                END ''begin 15''  $
@DP15.12.2                BEGIN
POSCMD,SYSTIME,PURGE,MIGRESET,LSFILTER,MIGTEST $
@DP15.12.2                ''POSITIONAL CMD,SYSTEM TIME,PURGE ALL TRACKS,MIG
RESET,
@DP15.12.2                LOW SPEED FILTER,MIG TEST TARGET CMD''
@DP15.12.2                IF IDSMHD(0,SUBCODE) EQ 2 AND IDSMHD(0,IFC) EQ 3
@DP15.12.2                ''PURGE ALL TRACKS COMMAND''
@DP15.12.2                THEN BEGIN $
@DP15.12.2                    SET IDSMPK(0,TASK) TO 4 $
@DP15.12.2                    SET IDSMPK(0,MODULE) TO SMFRIGMN $
@DP15.12.2                END ''PURGE''  $
@DP15.12.2                ELSE
@DP15.12.2                    SET FNDIT TO TRUE $
@DP15.12.2                END $
@DP15.12.2                BEGIN MOD4CNTL,DECOYS $
@DP15.12.2                SET FNDIT TO TRUE $

```

```

@DP15.12.2      END $
@DP15.12.2      END ''FOR''$
@DP15.12.2      END ''RDRCMDS'' $
@DP15.12.2      BEGIN RDRMSG '' radar select command''$
@DP15.12.2      SET IDSMPK(0,TASK) TO 6 $
@DP15.12.2      SET IDSMPK(0,MODULE) TO SMFRIGMN $
@DP15.12.2      END $
@DP15.12.2END   ''FOR'' $

```

SMA

Discrepancy Report

Missing:

```

SMA4 -> SMA8 DMUREAD 280
SMA4 -> SMG5 DMUREAD 280
SMA5 -> SMA4 DMUREAD 280
SMA7 -> SMA7 REMOVE
SMA8 -> SMA7 REMOVE
SMA9 -> SMA8 DMUREAD 280
SMA9 -> SMG5 DMUREAD 280

```

Extra:

```

SMA4
SMA4 -> SMG6 DMUWRITE 283
SMA4 -> SMG4 DMUREAD 280
SMA5 -> SMG5 DMUREAD 283
SMA5 -> SMG5 DMUREAD 283
SMA6 -> SMA6 DMUWRITE 283
SMA9 -> SMG5 DMUREAD 283

```

• SMA9 -> SMA8 DMUREAD 280 missing because the slice doesn't deal with conditionals correctly. The relevant code is:

```

SMAGEN.2050 (SMARDSCN):
  IF GVMODE EQ MODIFYSM
  OR GVMODE EQ PRINTSM
  THEN BEGIN  ''READ SCENARIO SMAGEN''$
    SET IDDMSRD(0,MODULE) TO SMAGENMN  ''SMAGEN MODULE''$
    SET IDDMSRD(0,TASK) TO SMARDCTN
  END        ''READ SCENARIO SMAGEN''$

```



```

ELSE BEGIN  ''LOAD SCENARIO SMGSWP''$
    SET IDDMSRD(0,MODULE) TO SMGSWPMN  ''SMGSWP MODULE''$
    SET IDDMSRD(0,TASK) TO SMGRDCTN
END      ''LOAD SCENARIO SMGSWP''$

```

- SMA4, SMA5, SMA9 -> SMG5 DMUREAD 280 is missing, and file number was found. As we can see this for statement indicates that 280, 281, 282, and 283 is possible, so both the documentation and the automatic process is wrong. If the conditionals are included in the slice, then the automatic method should be “more” correct.

```

SMAGEN.2062 (SMARDSCN):
    FOR GVSCENIX  ''FOR SCENARIO INDEX'' $
@DP15.1 BEGIN 0,1,2      ''SCENARIO FILE A''$
    SET IDDMSRD(0,FILEID) TO MNSMSCNA
    SET IDDMSRD(0,STARTREC) TO SMLSCEN*GVSCENIX + RSCNDIR
END  ''SCENARIO FILE A''$
@DP15.1 BEGIN 3,4,5      ''SCENARIO FILE B''$
    SET IDDMSRD(0,FILEID) TO MNSMSCNB ''SCENARIO FILE B''$
@DP15.1  SET IDDMSRD(0,STARTREC) TO SMLSCEN*(GVSCENIX-3)'
END  ''SCENARIO FILE B''$
@DP15.1 BEGIN 6,7,8      ''SCENARIO FILE C''$
    SET IDDMSRD(0,FILEID) TO MNSMSCNC ''SCENARIO FILE C''$
@DP15.1  SET IDDMSRD(0,STARTREC) TO SMLSCEN*(GVSCENIX-6)
END  ''SCENARIO FILE C''$
@DP15.1 BEGIN 9,10      ''SCENARIO FILE D''$
@DP15.1  SET IDDMSRD(0,FILEID) TO MNSMSCND ''SCENARIO FILE D''$
@DP15.1  SET IDDMSRD(0,STARTREC) TO SMLSCEN*(GVSCENIX-9)
@DP15.1 END  ''SCENARIO FILE D''$

```

- SMA7 -> SMA7 (REMOVE) and SMA7 -> SMA8 (REMOVE) are missing. The only remove call is the one listed below, yet we don't know how to handle a call where the whole table isn't passed to RTOS.

```

SMAEGN.1837 (SMAPRTOP):
    RTOS  INPUT REMOVE,CORAD(IDREGTSK(0,MODULE))
    OUTPUT DUMMY ''HAVE RTOS DESCHEDULE

```

- SMA5 -> SMG5 DMUREAD 283 is found, but not in the documentation. SMABUILD (SMA5) calls SMARDSCN (SMA9) which tasks SMG5 via a DMUREAD call. The relevant code is:

```

SMAGEN.2058 (SMARDSCN)
    SET IDDMSRD(0,MODULE) TO SMGSWPMN
SMAGEN.2059 (SMARDSCN)
    SET IDDMSRD(0,TASK) TO SMGRDCTN
SMAGEN.2093 (SMARDSCN)
    DMUREAD INPUT CORAD(IDDMSRD) OUTPUT...

```

- SMA5 -> SMA4 DMUREAD 280 is missing, because of the problems with conditionals. The relevant code is:

```
SMAGEN.2053 (SMARDSCN)
    SET IDDMSRD(0, MODULE) TO SMAGENMN
SMAGEN.2054 (SMARDSCN)
    SET IDDMSRD(0, TASK) TO SMARDCTN
SMAGEN.2058 (SMARDSCN)
    SET IDDMSRD(0, MODULE) TO SMGSWPMN
SMAGEN.2059 (SMARDSCN)
    SET IDDMSRD(0, TASK) TO SMGRDCTN
SMAGEN.2093 (SMARDSCN)
    DMUREAD INPUT CORAD(IDDMSRD) OUTPUT...
```

- SMA4 -> ? was found. The following code indicates that the RTOS call is made without setting the module and task number in that task. This is an example where the data dependencies cannot be found via this static method

```
SMAGEN.1236 (SMAGENT)
    SMAILSA
SMAGEN.2572 (SMAILSA)
    RTOS INPUT REGISTER, CORAD(IDREGDSC)...
```

- SMA4 -> SMA6 DMUWRITE 283 was found but not in the manually generated documentation. Conditionalized code is again the culprit. As we can see this for statement indicates that 280, 281, 282, and 283 is possible, so both the documentation and the automatic process is wrong. If the conditionals are included in the slice, then the automatic method should be "more" correct.

```
SMAGEN.1630 (SMARECRD)
    SET IDDMSWR(0, FILEID) TO MNSMSCNA      ( = 280)
SMAGEN.1630 (SMARECRD)
    SET IDDMSWR(0, FILEID) TO MNSMSCNB      ( = 281)
SMAGEN.1630 (SMARECRD)
    SET IDDMSWR(0, FILEID) TO MNSMSCNC      ( = 282)
SMAGEN.1630 (SMARECRD)
    SET IDDMSWR(0, FILEID) TO MNSMSCND      ( = 283)
SMAGEN.1653 (SMARECRD)
    DMUWRITE INPUT CORAD(IDDMSWR)....
```

SMB

No discrepancies

SMD

No discrepancies

SME

This module was run interactively.

Discrepancy Report

Missing:

SME4 -> SME4 DELAY 625 msec
SME4 -> IOREQUES 567
SME5 -> SME4 DELAY 625 msec
SME5 -> IOREQUES 567
SME6 -> SME4 DELAY 625 msec
SME6 -> IOREQUES 567
SME7 -> SME4 DELAY 625 msec
SME7 -> IOREQUES 567

Extra:

SME4 -> IOREQUES 568
SME4 -> IOREQUES 603
SME5 -> IOREQUES 568
SME5 -> IOREQUES 603
SME6 -> IOREQUES 568
SME6 -> IOREQUES 603
SME7 -> IOREQUES 568
SME7 -> IOREQUES 603

; Missing

SMF

Discrepancy Report

Missing:

SMF5 -> SME5
SMF5 -> SME6
SMF5 -> SME7
SMF5 -> SMF8 delay 10 sec
SMF5 -> SMF8 REMOVE
SMF5 -> SMF9 delay 10 sec
SMF5 -> SMF9 REMOVE
SMF5 -> SMF10 delay 10 sec
SMF5 -> SMF10 REMOVE
SMF6 -> SCB18
SMF6 IOREQUES 589
SMF7 IOREQUES 589
SMF8 IOREQUES 589
SMF9 IOREQUES 589
SMF10 IOREQUES 589

Extra:

SMF4 IOREQUES 567
SMF5 IOREQUES 567
SMF6 IOREQUES 567
SMF7 IOREQUES 567
SMF8 IOREQUES 567
SMF9 IOREQUES 567
SMF10 IOREQUES 567

- SMF5 -> SMF8,9,10 REMOVE are missing because RDRINDX is not set, TASKNUM always 7. The value of RDRINDX is equal to SMFRDRID(0, RADX), which is not set to any value within the module. The relevant code is:

```
SMF.6615 (SMFRATE)
      SET IDSMFRIG(0,TASKNUM) TO RDRINDX + 7
SMF.5138 (SMFRDM)
      SET RDRINDX TO SMFRDRID(0,RADX)
```

- SMF5 -> SME5,6,7: are missing Because RDRINDX not set, TASKNUM always 4. The value of RDRINDX is equal to SMFRDRID(0, RADX), which is not set to any value within the module.

```

SMF.6366 (SMFSCAN)
    SET TASKNUM TO RDRINDX + 4
SMF.5138 (SMFRDM)
    SET RDRINDX TO SMFRDRID(0,RADX)

```

- SMF5 -> SMF5,6,7,8 should have a delay Seems like there should always be a delay...

```

;    SET DELAY TO 20 "to delay 1st execution by 10 seconds    "$
;    SET MODNUM TO SMFRIGMN
;    SET TASKNUM TO RDRINDX + 7

```

- SMF4,5,6,7,8,9,10 IOREQUES 567 should be 568-571, but we don't evaluate loops (related to missing 568). The relevent code is:

```

;
VARY VSUINDX FROM 0 THRU SMNRDR - 1
  IF GTMOCU(VSUINDX,RDRNO) EQ RADARIFF THEN ''VSU sim'ing Rdr''
  BEGIN
    SET IDRIOPKT(0,BUSADDR) TO VSUINDX + 567 ''Set Logical
    RTOS INPUT IOREQUES, CORAD(IDRIOPKT) OUTPUT DUMMY
  END
END ''VARY Loop''

```

- SMF6,7,8,9,10 IOREQUES 567 should be 589, (or something else depending upon what IDFRECV(0,GVOCUNUM) is, which wasn't in slice for some reason) (related to missing 589)

```

;
(SMFSLM)
  SET OCUNUMBR TO IDFRECV(0,GVOCUNUM)
(SMFSLM)
  SET VSUINDEX TO (OCUNUMBR - 16) / 4
(SMFNORTH) SET BUSADR TO 589
;

IF VSUIO EQ 0 THEN ''Output to go to <SIC>''
  SET IDRIOPKT(0,BUSADDR) TO BUSADR
ELSE ''Output to go to a single VSU''
  SET IDRIOPKT(0,BUSADDR) TO VSUINDEX + 567 ''Set Logical
  RTOS INPUT IOREQUES, CORAD(IDRIOPKT) OUTPUT DUMMY

```

SMG

Discrepancy Report

Missing:

SMG04 -> SMG05 DMUREAD 280

Extra:

SMG04 -> SMG04 DMUREAD 280

- SMG4 -> SMG5 is missing but SMG4 -> SMG4 was found instead. The relevant code is:

```
SMGPNN.308: SWACTNTN EQUALS 4
```

```
SMGBWP.2581 (SMGXXSIM):
```

```
SET IDDMSRD(0,TASK) TO SWACTNTN
```

```
SMGBWP.2584 (SMGXXSIM):
```

```
DMUREAD INPUT CORAD(IDDMSRD) OUTPUT DMSCODE, DMSADDR
```

SRC

Discrepancy Report

Missing:

SRC6 -> IOREQUES 1940

SRC12 -> IOREQUES 1940

SRD

Discrepancy Report

Extra:

SRD8 -> DMUDISTR 52

- SRD8 -> DMUDISTR 52 (CNGATAOG) is found, but not in the documentation. In what follow, the procedure SRDSIPT is the entry procedure of task 8. The relevant code is:

```
SDRID.14881 (SRDSIPT):  
    SET IDDMUPR(0, FILEID) TO CNFATAOG  
SRDID.14884 (SRDSIPT)  
    DMUDISTR INPUT CORAD(IDDMUPR) OUTPUT GCOMPCode, GDMADDR
```

SRE

Discrepancy Report

Missing:

SRE4 -> DSC4

SRE5 -> DSC4

SRE6 -> DSC4

- All of these missing events are due to missing conditional code in the program slices. The relevant code is:

```
SRETE.3871 (SREDSOUT)  
    SET IDDSREQ(0, TSKNUM) TO 5  
SRETE.3874 (SREDSOUT)  
    SET IDDSREQ(0, TSKNUM) TO 4  
SRETE.3879 (SREDSOUT)  
    RTOS INPUT REGISTER, CORAD(IDDSREQ) OUTPUT DUMMY
```

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.